INFORMATIK-GRUNDLAGEN

Programmieren

Der persönliche Software-Prozess (PSP)

Fachhochschule Fulda
Fachbereich Elektrotechnik
Prof. Dr. Timm Grams

Datei: programmieren.doc
15. Februar 2010
(Erstausgabe: 20.12.2001)

Hinweise

Der gesamte Kurs wird von einem Leitgedanken beherrscht: Programmieren ist Handwerk, Kunst und Wissenschaft. Gegenstand der Lehrveranstaltung ist der *persönliche Software-Prozess*. Im Zentrum dieser Lehrveranstaltung steht das *Programmieren im Kleinen,* also das Erstellen von Programmen, die einen Algorithmus, eine Funktion, eine Methode oder dergleichen realisieren und deren Texte auf eine Seite passen. Beispiele sind in C programmiert. Hauptaugenmerk gilt dem fehlerfreien Programm und dem Weg dorthin:

- Programmieren und Testen nach Regeln
- Regelkreis des selbstkontrollierten Programmierens
- Beweisgeleitetes Programmieren: das *Denken vom Resultat her*
- Kontrolle von Rundungs- und Verfahrensfehlern

In der letzten Lektion werden an einem einfachen Beispiel verschiedene Programmierstile vorgeführt. Diese Lektion ist ein Ausblick auf höhere Sprachelemente, wie sie die objektorientierten Sprachen zur Verfügung stellen.

Zur Typographie: Programme und insbesondere deren Variablen werden grundsätzlich nicht kursiv geschrieben. Kursiv stehen alle Variablen und Funktionsbezeichner, die nicht Bezeichner in einem Programm sind, also insbesondere die Variablen im mathematischen Sinn oder Abkürzungen und Bezeichner für Programmteile. (`Schreibmaschinenschrift` wird verwendet, wenn diese Schrift der Übersichtlichkeit dient: Verbesserung der Unterscheidung zwischen Programmteilen und beschreibendem Text, Verdeutlichung der Programmstruktur durch Einrücken und Unterstützung der Lesbarkeit bei Umformungen durch geeignete Positionierung äquivalenter Ausdrücke in aufeinanderfolgenden Textzeilen.)

Optionale Aufgaben und Abschnitte sind mit Sternchen gekennzeichnet. Welche der Aufgaben zu bearbeiten sind, wird erst im Laufe des Kurses festgelegt. Zwei oder mehr Sternchen weisen auf besonders anspruchsvolle Aufgaben oder Abschnitte hin.

Begleitmaterial: Das Skriptum ist gedacht als Leitfaden der Lehrveranstaltung; es ist *nicht* gedacht als Lehrbuch. Bücher, die das Thema des Kurses - nämlich das Programmierung im Kleinen und den persönlichen Software-Prozess - am ehesten treffen, sind Humphreys *Introduction to the Personal Software Process* (ab Kapitel 11) und mein *Denkfallen und Programmierfehler*. Meine Denkfallen-Seiten im Web bringen Hintergrundmaterial und Hinweise auf weiterführende Literatur:

```
http://www.fh-fulda.de/~grams/dnkfln.htm
http://www.fh-fulda.de/~grams/Denkfallen/SystemHaupt.htm
```

Vorkenntnisse: Der Stoff der Lehrveranstaltung Informatik 1 und 2 wird vorausgesetzt. Insbesondere: Aussagenlogik, Syntax und Semantik der Programmiersprache C, Fließpunktdarstellung reeller Zahlen, ANSI/IEEE Standard 754-1985, Abschätzung von Rundungsfehlern.

Leistungsnachweis: Jeder Lektion sind Programmieraufgaben zugeordnet, die jeder Teilnehmer auf sich allein gestellt lösen soll. In dieser Lehrveranstaltung ist Gruppenarbeit ausnahmsweise nicht die Regel! Die Erledigung der Aufgaben wird durch jeden Teilnehmer auf *einem DIN-A4-Blatt* (maximal 2 Seiten) dokumentiert (Name und Datum nicht vergessen!). Dieses Dokument enthält das Programm oder relevante Ausschnitte daraus, die Kommentierung des Ergebnisses sowie die Kurzdarstellung der gewonnenen Erkenntnisse. Bei Nachfrage sind die zugehörigen Programme vorzuführen. Die Erledigung der Aufgaben wird durch Testat auf dem Dokument bestätigt. Der Leistungsnachweis wird in Fachgesprächen erbracht. Die Testate sind dabei Beurteilungsgrundlage.

Gliederung

Literaturverzeichnis

Programmiersprache

Böttcher, A.; Kneißl, F.: Informatik für Ingenieure. Grundlagen und Programmierung in C. Oldenbourg, München, Wien 1999

Capper, D. M.: „C++ for Scientists, Engineers and Mathematicians“, Springer-Verlag, London 1994. *Enthält Gleitpunkt-Darstellung reeller Zahlen nach ANSI/IEEE 754 (S. 399-403)*

Kernighan, B. W.; Ritchie, D. M.: The C Programming Language. 2nd edition. Prentice Hall, Englewood Cliffs, New Jersey 1988

Stroustrup, B.: The C++ Programming Language. 2nd Ed. Addison-Wesley, Reading, Mass. 1995

Programmier- und Testmethodik

Alagic, S.; Arbib, M. A.: The Design of Well-Structured and Correct Programs. Springer, New York 1978

Dahl, O.-J.; Dijkstra, E. W.; Hoare, C. A. R.: Structured Programming. Academic Press, London 1972. *Eine Sammlung der klassischen Beiträge dieser Software-Pioniere.*

Grams, T.: Denkfallen und Programmierfehler. Springer, Berlin, Heidelberg 1990

Humphrey, W. S.: Introduction to the Personal Software Process. Addison-Wesley, Reading, Mass. 1997

Linger, R. C.; Mills, H. D.; Witt, B. I.: Structured Programming. Theory and Practice. Addison-Wesley, Reading, Mass. 1979

Myers, G. J.: Methodisches Testen von Programmen. Oldenbourg Verlag, München 1987

Beweisgeleitetes Programmieren

Baber, R. L.: Fehlerfreie Programmierung für den Software-Zauberlehrling. Oldenbourg, München, Wien 1990

Gries, D.: The Science of Programming. Springer, Heidelberg 1981

Numerik

ANSI/IEEE Std. 754-1985: A Standard for Binary Floating-Point Arithmetic. New York 1985. *Definiert die Datentypen* `float` *und* `double`.

ANSI/IEEE Std. 854-1987: A Standard for Radix-Independent Floating-Point Arithmetic. New York 1987

Engeln-Müllges, G.; Uhlig, F.: Numerical Algorithms with C. Springer, Berlin, Heidelberg 1996

Hammer, R.; Hocks, M.; Kulisch, U.; Ratz, D.: C++ Toolbox for Verified Computing. Basic Numerical Problems. Springer, Berlin, Heidelberg 1995

Hämmerlin, G.; Hoffmann, K.-H.: Numerische Mathematik. Springer, Berlin, Heidelberg 1992

Knuth, D.: The Art of Computer Programming. Vol. 2. Seminumerical Algorithms. Addison-Wesley 1981

Schwetlick, H.; Kretzschmar, H.: Numerische Verfahren für Naturwissenschaftler und Ingenieure. Fachbuchverlag, Leipzig 1991

Stoer, J.: Numerische Mathematik 1. Springer, Berlin, Heidelberg 1994

Stoer, J.; Bulirsch, R.: Numerische Mathematik 2. Springer-Verlag, Berlin, Heidelberg 1990

Weitere Quellen und wichtige Übersichtswerke

Aho, A. V.; Hopcroft, J. E.; Ullman, J. D.: Data Structures and Algorithms. Addison-Wesley, Reading, Massachusetts 1983

Broy, M.; Denert, E. (Eds.): Software Pioneers. Contributions to Software Engineering. Springer-Verlag, Berlin, Heidelberg 2002. *Tagungsbericht der Tagung der Software-Pioniere mit aktuellen Beiträgen und historischen Dokumenten. (Teilnehmer u. a.: Friedrich L. Bauer, Ole-Johan Dahl, Niklaus Wirth, Edsger W. Dijkstra, C. A. R. Hoare, David L. Parnas, Tom De Marco, Barry Boehm, Erich Gamma)*

Feijen, W. H. J.; van Gasteren, A. J. M.; Gries, D.; Misra, J.: Beauty is Our Business. Springer, New York 1990. *Ein Schatzkästchen! Viele gute Ideen in 54 Einzelbeiträgen von jeweils nur wenigen Seiten prägnant dargestellt von den Größen der Informatik und anderen.*

Fellner, W. D.: Computer Grafik. Bibliographisches Institut, Mannheim 1988

Grams, T.: Grundlagen des Qualitäts- und Risikomanagements - Zuverlässigkeit, Sicherheit, Bedienbarkeit. Vieweg Praxiswissen, Braunschweig, Wiesbaden 2001

Knuth, D.: The Art of Computer Programming. Vol. 1. Fundamental Algorithms. Addison-Wesley 1973

Lüneburg, H.: Leonardi Pisani Liber Abbaci oder Lesevergnügen eines Mathematikers. BI Wissenschaftsverlage, Mannheim 1992

Wirth, N.: Drawing Lines, Circles, and Ellipses in a Raster. Aus: Beauty is Our Business (Feijen u. a., 1990, 427-434)

1 Programmierübungen zum Einstieg

Thema der Lehrveranstaltung ist das *Programmieren im Kleinen*. In der ersten Lektion geht es vor allem darum, festzustellen, welche Reife der *persönliche Software-Prozess* bei den Teilnehmern bereits hat und auch, wo er noch mangelhaft sind.

Die Aufgaben dieser Lektion werden von jedem Teilnehmer in *Einzelarbeit* und in Klausur gleich zu Beginn der Lehrveranstaltung bearbeitet. Die Programme werden auf einer Diskette gesammelt. Zur Unterscheidung wird die Datei eines Programms mit dem Aufgabentitel und dem Nachnamen des Autors benannt. Mein C-Quelltext der ersten Programmierübung heißt also `GeradeGrams.c`; falls es einen Jürgen Müller und einen Holger Müller im Kurs gibt, machen wir das so: `GeradeMuellerJ.c` bzw. `GeradeMuellerH.c`.

Vorbemerkung: Die booleschen Konstanten `false` und `true` deklarieren wir folgendermaßen:

```
typedef enum {false, true} bool;
```

Aufgaben

1.1 *Gerade:* Gegeben sind drei Punkte in der Ebene: A, B und C. Die Funktion mit der Deklaration `bool gerade(int alpha)` hat als Parameter Alpha den Winkel ∠ABC. Sie soll genau dann den Wert `true` ausgeben, wenn die drei Punkte auf einer Geraden liegen und ansonsten den Wert `false`.

1.2 *Parkplatz:* Auf einem Parkplatz stehen *n* Fahrzeuge: Motorräder mit je zwei und PKWs mit je vier Rädern. Insgesamt sind es *m* Räder. Wie viele PKWs und wie viele Motorräder sind auf dem Platz? Schreiben Sie ein Programm, das die Werte *n* und *m* entgegennimmt und die Anzahl der PKWs und die Anzahl der Motorräder ausgibt. Falls das Programm nach einem Compiliervorgang nicht oder nicht zufriedenstellend arbeitet, dokumentieren Sie die daraufhin notwendigen Verbesserungen im Programm, indem sie den alten Programmtext auskommentieren und eine Begründung für die Änderung hinzufügen.

1.3 *Zweierpotenz:* Gesucht ist die größte Zweierpotenz, die nicht größer als eine vorgegebene Zahl *z* ist. Schreiben Sie ein Programm, das diese Aufgabe löst. Alle Zahlen sind ganz. Alle Daten sind vom Typ `int` oder `long`.

1.4 *Dreiecke:* Gegeben seien die ganzen Zahlen *a*, *b* und *c* (Eingabe). Die Zahlen werden als Seitenlängen eines Dreiecks interpretiert. Schreiben Sie ein Programm, das feststellt, ob es sich um ein ungleichseitiges, ein gleichschenkliges, oder um ein gleichseitiges Dreieck handelt (Ausgabe).

1.5 *Summe:* Schreiben Sie ein Programm zur Ermittlung eines möglichst guten Näherungswertes für die Summe $\sum_{k=1}^{\infty} \frac{1}{k^2}$.

1.6 *Nullspalte:* Schreiben Sie den Rumpf zur Funktion mit dem Kopf

```
bool nullspalte(int a[m][n])
```

Die Funktion soll genau dann den Wert `true` ausgeben, wenn die Matrix `a[m][n]` eine Spalte mit lauter Nullen enthält. (Die Namen m und n sind mittels #define-Anweisung durch Zahlenkonstante zu ersetzen.)

Denkfallen

Wir haben Schwierigkeiten, auf Anhieb korrekte Programme zu schreiben. Oft stecken Denkfallen hinter diesen Schwierigkeiten. Eine *Denkfalle* ist gegeben, wenn eine Problemsituation einen weit verbreiteten und bewährten Denkmechanismus in Gang setzt, und wenn dieser Denkmechanismus mit der gegebenen Situation nicht zurechtkommt und mit hoher Wahrscheinlichkeit zum Irrtum führt.

Weit verbreitet und bewährt heißt, dass dieser Denkmechanismus zum *Hintergrundwissen* einer ganzen Gruppe von Personen - beispielsweise der Gemeinde der Programmierer oder der Angehörigen einer Zivilisation - gehört. Folglich werden Fehler aufgrund von Denkfallen in dieser Population mit hoher Wahrscheinlichkeit und immer wieder begangen. Und diese Fehler entziehen sich den üblichen Kontrollmechanismen, wie beispielsweise dem erneuten Durchlesen eines Programmtexts durch dessen Urheber oder durch andere.

Denkfallen gehen offenbar auf an sich nützliche Denkmechanismen zurück. Das kann gar nicht anders sein. Denn wären sie nicht nützlich, könnten sie gar nicht weit verbreitet sein. Die mit ihnen ausgestatteten Populationen hätten im Überlebenskampf keine Chance und wären längst untergegangen.

Ich lege die folgende Taxonomie der Denkfallen zu Grunde (Grams 1990, 2001)[1]:

1. Übergeordnete Prinzipien

- *Scheinwerferprinzip*

Aus dem riesigen Informationsangebot der Außenwelt werden - wegen der Kapazitätsbeschränkung unseres Kurzzeitgedächtnisses und des damit zusammenhängenden *Engpasses der Wahrnehmung* - nur relativ kleine Portionen ausgewählt und bewusst verarbeitet. Die Auswahl und Filterung der Information hängt von der Ausrichtung des „Scheinwerfers der Aufmerksamkeit“ ab. Karl Raimund Popper spricht vom *Scheinwerfermodell der Erkenntnis*: „Wir erfahren ja erst aus den Hypothesen, für welche Beobachtungen wir uns interessieren sollen, welche Beobachtungen wir machen sollen“.

- *Sparsamkeitsprinzip*

Das Sparsamkeits- oder Ökonomieprinzip besagt, dass Arten und Individuen, die ökonomisch mit den Ressourcen umgehen, Vorteile im Konkurrenzkampf haben. Durch Auslese wird folglich der Aufwand zur Erreichung eines bestimmten Zwecks minimiert. Bei übertriebener Anwendung des Sparsamkeitsprinzips können wir *Wesentliches übersehen*. Wir denken zu einfach.

2. Die „angeborenen Lehrmeister„

Für Konrad Lorenz sind „die angeborenen Lehrmeister ... dasjenige, was vor allem Lernen da ist und da sein muss, um Lernen möglich zu machen“.

- *Strukturerwartung*

Alles Leben geht augenscheinlich von der Hypothese eines objektiv existierenden Kosmos aus, der von Recht und Ordnung zusammengehalten wird. Die Strukturerwartung wirkt sich bei der optischen Wahrnehmung als *Prägnanztendenz* aus. Das ist die Extraktion und Verstärkung wesentlicher Merkmale. Man spricht auch von der *Sinnsuche des Wahrnehmungsapparats*. Strukturerwartung und Prägnanztendenz schießen zuweilen über das Ziel hinaus; dann kann es zu einer *Überschätzung des Ordnungsgehalts* der Dinge kommen. Beispiele: der Necker-Würfel und die „rutschende Leiter“.

- *Kausalitätserwartung*

Nach Rupert Riedl enthält die „Hypothese von der Ursache die Erwartung, dass Gleiches dieselbe Ursache haben werde. Dies ist zunächst nicht mehr als ein Urteil im Voraus. Aber dieses Vorurteil bewährt sich... in einem derartigen Übermaß an Fällen, dass es jedem im Prinzipe andersartigen Urteil oder dem Urteils-Verzicht überlegen ist“. Verhängnisvoll wird das Prinzip bei ausschließlich *linearem Ursache-Wirkungs-Denken* (linear cause-effect thinking), und wenn wir die Vernetzung der Ursachen und die Nebenwirkungen unserer Handlungen außer Acht lassen.

- *Anlage zur Induktion*

Unsere Anlage zur Induktion, also unser Hang zu Erweiterungsschlüssen, arbeitet nach folgendem Argu-

---

[1] Diese Taxonomie ist auf der Web-Page "http://www.fh-fulda.de/~grams/Denkfallen/SystemHaupt.htm" weiter ausgeführt und erläutert. Von dort kommt man auch zu meiner Denkfallenseite, die typische Beispiele bringt.

mentationsmuster: Wenn sich aus der Theorie (Hypothese) H ein Ereignis E vorhersagen lässt, und wenn gleichzeitig das Ereignis E aufgrund des bisherigen Wissens recht unwahrscheinlich ist, dann wird die Theorie H aufgrund einer Beobachtung des Ereignisses E glaubwürdiger. Kurz: Aus „H impliziert E“ und „E ist wahr“ folgt „H wird glaubwürdiger“. Diese Art des plausiblen Schließens zusammen mit dem linearen Ursache-Wirkungs-Denken (Kausalitätserwartung) macht generalisierende Aussagen überhaupt erst möglich. So kommen wir zu wissenschaftlichen Hypothesen und schließlich Theorien. Wir tendieren dazu, Induktionsschlüsse fälschlich wie logische Schlussfolgerungen zu interpretieren. Wir unterscheiden nicht konsequent genug zwischen „Aus H folgt E“ und „Aus E folgt H“. Damit einher geht die *Überbewertung bestätigender Information* (confirmation bias).

3. Bedingungen des Denkens

- *Assoziationen*

Assoziationen, also die Verknüpfung von zunächst unabhängig voneinander funktionierenden Nervenvorgängen, sind es, die das Erkennen und Abspeichern von Zusammenhängen und Gesetzmäßigkeiten möglich machen. Für das Denken von größter Bedeutung ist, dass neue Denkinhalte in ein Netz von miteinander verbundenen (assoziierten) Informationen eingebettet werden und dass bei Aktivierung eines solchen Denkinhalts das assoziative Umfeld mit aktiviert wird. Dadurch entdecken wir Zusammenhänge, können wir Schlussfolgerungen ziehen. Aber es kommt gelegentlich vor, dass falsche oder irreführende Assoziationen geweckt werden.

- *Einstellungen*

Die Automatisierung der Denkvorgänge - auch *Einstellungseffekt (mind-set)* genannt - beruht darauf, dass uns frühere Erfahrungen dazu verleiten, beim Lösen eines Problems bestimmte Denk- und Handlungsweisen (Operatoren) gegenüber anderen vorzuziehen (Anderson, 1988, S. 210 ff.). Das blinde Wiederholen von früher erworbenen Reaktionsmustern entlastet den Denkapparat. Es kann aber auch das Lösen von Problemen erschweren. Es besteht die Tendenz, in einen Zustand der Mechanisierung - oder: *Einstellung* - zu verfallen. Zu Fehlern kommt es, wenn der Zustand nicht verlassen wird, obwohl dies angezeigt ist.

Die bei der Lösung der Aufgaben aufgetretenen Programmierfehler werden unter Zuhilfenahme dieser Taxonomie analysiert. In den folgenden Lektionen geht es auch um die *Programmiermethoden*, mit denen wir diese Denkfallen vermeiden können.

2 Der persönliche Software-Prozess (PSP)

Vorgehensmodell des Programmierens

Wir sind schnelle Entscheider. Diese Optimierung unseres Verhaltens haben wir vermutlich unseren Vorfahren zu verdanken und von ihnen geerbt.

Für sie war es lebenswichtig, schnell zu erkennen, ob das, was sich da im Gebüsch bewegt ein gefährliches Raubtier ist, oder eine harmlose Beute. Dementsprechend galt es meist, rasch zu handeln.

Aber gerade das manchmal vorschnelle Entscheiden zusammen mit dem Scheinwerferprinzip und dem Festhalten an einmal gefassten Urteilen und Hypothesen (Induktion) ist es, was dem Programmierer das Leben schwer macht. Geht er dieser Neigung nach, fabriziert er Fehler, die sich im Programm verfestigen, die es undurchsichtig machen, und die in späteren Phasen viel Zeit, Nerven und Geld kosten.

Für ihn zahlt sich eher das ruhige Durchdenken einer Problemlage und das Abwägen aus. Auch für den *persönlichen Software-Prozess* ist in Anlehnung an das Lebenszyklusmodell - ein mehrstufiges Vorgehen empfehlenswert. Es besteht aus *klar voneinander getrennten Arbeitsabschnitten, die in einer bestimmten Abfolge stehen* (Bild 2.1).

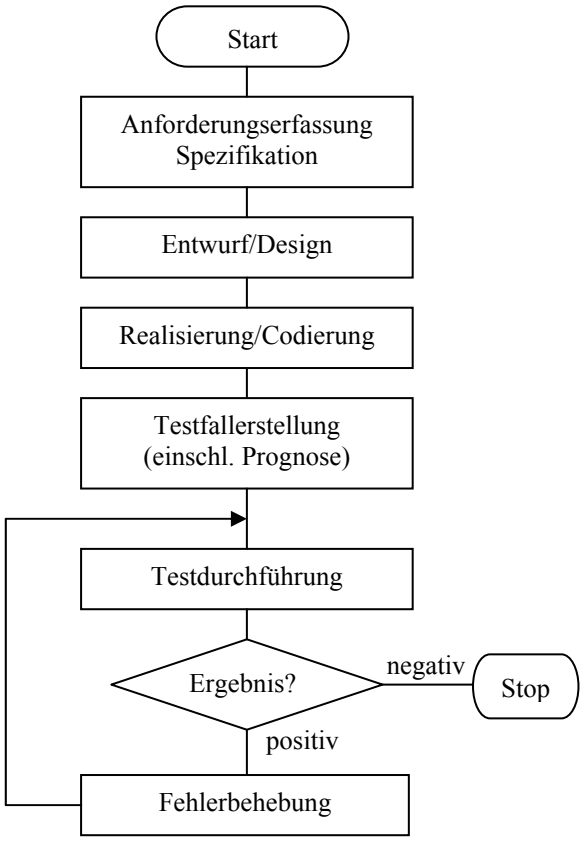


Bild 2.1 Ablaufplan des Programmierens

Die negative Methode und das Testen nach Regeln

Die *negative Methode* soll speziell Denkfallen aufdecken und überwinden helfen. Die negative Methode umfasst die folgenden allgemeinen Verhaltensregeln:

1. Warnzeichen für die Unangemessenheit der eigenen Gewohnheiten, Rezepte und Methoden suchen und ernst nehmen.
2. Nicht nach Bestätigung von Vorurteilen und Hypothesen suchen, sondern
3. vor allem nach Gegenbeispielen und Widerlegungen Ausschau halten.
4. Die Ursachen von Fehlern gründlich analysieren.
5. Die Methoden und Techniken der Problemlösung - also die eigene Kompetenz - weiterentwickeln, so dass diese Fehler künftig vermieden werden.

Vor allem beim Testen sollte die negative Methode angewendet werden.

Sei *T* ein korrekt entworfener Test. Das heißt: Jedes nach der gegebenen Spezifikation korrekt entworfene Programm möge den Test bestehen. Mit P_T wird die Wahrscheinlichkeit bezeichnet, dass das Programm diesen Test besteht. Ein Test ist um so wirksamer, je kleiner P_T ist. Sei k die Korrektheitswahrscheinlichkeit vor dem Test und k_T die Korrektheitswahrscheinlichkeit nach Bestehen des Tests. Das Verhältnis k_T/k nennen wir *Bewährungsgrad* durch den Test *T*. Die Regeln der Wahrscheinlichkeitsrechnung liefern die Formel $\frac{k_T}{k} = \frac{1}{P_T}$. Wenn ein

System den Test nur mit der Wahrscheinlichkeit von 10% besteht, dann steigt die Korrektheitswahrscheinlichkeit durch den bestandenen Test um den Faktor zehn. Auch für die Ermittlung von P_T muss man auf Schätzverfahren und Erfahrungswissen zurückgreifen. Jedenfalls ist es leichter, etwas über die Wirksamkeit eines Tests und damit über den Zuwachs der Korrektheitswahrscheinlichkeiten (k_T/k) zu erfahren, als über die Korrektheitswahrscheinlichkeiten selbst (k_T oder k). Aggressive Tests zeichnen die negative Methode aus. Ziel ist eine möglichst große Wirksamkeit, also ein möglichst kleines P_T.

Die rationale Einstellung gegenüber Programmen - auch den eigenen - ist grundsätzliches Misstrauen. Aber es ist gar nicht so leicht, diese negative Methode gegen das eigene Programm zu wenden. Deshalb wird im Allgemeinen empfohlen, die Programme nicht durch den Programmierer selbst testen zu lassen. Hier geht es aber um den persönlichen Software-Prozess. Der Programmierer sollte alle Möglichkeiten zur Verbesserung seines Programms nutzen. Das Testen gehört dazu. Er muss es nur richtig angehen. Er muss sich einem gewissen Zwang aussetzen und sich an ein paar Regeln halten.

Die Trennung der Phasen Realisierung und Testfallerstellung (Bild 2.1) ist dabei von grundlegender Bedeutung. Sie geht mit einem *Wechsel der Einstellung* einher. Gries (1981, S. 169) drückt das so aus: „Der kluge Programmierer entwickelt sein Programm mit der Einstellung, dass ein korrektes Programm entwickelt werden kann und werden wird, vorausgesetzt, man wendet genügend Sorgfalt und Konzentration auf. Und dann testet er es gründlich mit der Einstellung, dass ein Fehler drin sein muss."

Die *Sinnsuche des Wahrnehmungsapparats*, unser Bestreben, Gesetzmäßigkeiten auch in irgendwelche eher unzusammenhängende Sachverhalte hineinzusehen, macht auch vor Testergebnissen nicht Halt. Fehler werden leicht übersehen, wenn man nicht bereits eine Vorstellung vom korrekten Ergebnis hat. Also gilt die Regel, dass vor dem eigentlichen Test sämtliche Testfälle einschließlich der erwarteten Ergebnisse formuliert und dokumentiert sein müssen. Die sorgfältig erstellte *Prognose* ist für die Wirksamkeit des Tests unerlässlich.

Die Fähigkeit, gute Testfälle zu erfinden, wächst mit der Erfahrung. Zur Unterstützung des Gedächtnisses empfiehlt es sich, einen *Regelkatalog für Testfälle* anzulegen und fortzuschreiben. Die folgenden Regeln haben sich allgemein bewährt:

1. Den *gesamten Eingabedatenraum* nach geeigneten Testfällen durchsuchen.
2. Gültige und ungültige Eingabedaten wählen.
3. Sonderfälle und Entartungen berücksichtigen.
4. Aus jeder *Äquivalenzklasse* wenigstens einen Eingabedatensatz vorsehen (äquivalent sind Datensätze dann, wenn das Programm vermutlich gleichartig darauf reagiert, wenn also zu erwarten ist, dass entweder bei jedem der Datensätze ein Fehler auftritt oder bei keinem). An die negativen („unnatürlichen") Zahlen denken!
5. *Grenzwerte* testen. Also vorzugsweise Werte vorgeben die an Intervallgrenzen (von Schleifen beispielsweise) liegen - diesseits und jenseits.
6. Elemente mit besonderen Eigenschaften wie 0 oder 1 berücksichtigen.
7. Die Extremwerte des Wertebereichs (größte Zahl, kleinste Zahl, größte negative Zahl, kleinste positive Zahl) in die Testfälle einbeziehen.

8. Die Testfälle samt Prognosen dokumentieren und in einer Datei aufheben, so dass der Test zu einem späteren Zeitpunkt - zum Beispiel nach einer Programmänderung im Rahmen eines *Regressionstests*[2] - mühelos wiederholt werden kann (Myers, 1987).

Aufgabe

2.1 Testen Sie Ihre Programme aus der ersten Lektion nach dem besprochenen Schema. Dokumentieren Sie kurz ihre Ergebnisse und Erkenntnisse.

[2] Das Wort Regression ist der Statistik entlehnt und meint soviel wie „das Zurückgehen von (auffällig großen oder kleinen) Merkmalswerten in Richtung Mittelwert“: Galton stellte fest, dass große Eltern zwar überdurchschnittlich große Kinder haben, dass diese Abweichungen aber in aufeinanderfolgenden Generationen zurückgehen (Székely, G. J.: Paradoxa. Klassische und neue Überraschungen aus Wahrscheinlichkeitsrechnung und mathematischer Statistik. Harri Deutsch, Thun, Frankfurt/M. 1990). In Analogie dazu sorgt ein Regressionstest dafür, dass die durch Programmänderungen eingeschleppten Fehler und die dadurch verursachten Abweichungen vom „Idealverhalten“ wieder eliminiert werden können.

3 Personenbezogene projektübergreifende Aktivitäten

Fehlerbuchführung - Defect Records

Wir betrachten die Fehler der Vergangenheit als einen teuer erworbener Wissensfundus. Sie werden in einem *Fehlerbuch* (Defect Record) festgehalten, das man besten auf einer Datei zugriffsbereit hält. Ich dokumentiere die lehrreichsten meiner Programmierfehler auch in den Dateien selbst, indem ich den entsprechenden Programmtext auskommentiere und das Stichwort „Denkfalle“ unmittelbar hinter das öffnende Kommentarzeichen setze. So lassen sich diese Fehler leicht wieder auffinden.

Ich empfehle, nur die typischen und lehrreichen Fehler in das Buch aufzunehmen. Ob ein Fehler in das Fehlerbuch gehört, sollte man davon abhängig machen, ob die Fehlersuche lange gedauert hat, die durch den Fehler verursachten Kosten hoch waren oder der Fehler lange unentdeckt geblieben ist.

Die Eintragung in das Fehlerbuch sollte wenigstens die folgenden Daten umfassen

- Kurzbezeichnung, Name des Fehlers
- Datum (wann entstanden, wann entdeckt)
- Programmname, Datum
- Fehlerkurzbeschreibung
- Ursache (Welche Denkfalle steckt dahinter?)
- Rückverfolgung (Ist der Fehlertyp neu? Wenn nicht: Warum ist er erneut aufgetreten?)
- Programmierregel, Gegenmaßnahme

Um es noch einmal zu betonen: Hier wird das Führen eines persönlichen Fehlerbuches empfohlen. Unabhängig davon werden im Rahmen von Software-Projekten Fehlerbücher geführt. Ein solches projektweites Fehlerbuch hat in erster Linie den Zweck, Programmierregeln für das Gesamtteam abzuleiten. Demgegenüber geht es beim persönlichen Fehlerbuch darum, den persönlichen Programmierstil zu verbessern.

Programmieren nach Regeln

Der Regelkreis des selbstkontrollierten Programmierens

Zweck des folgenden Regelkatalogs ist in erster Linie die Vermeidung von Programmierfehlern. Er ist weitgehend meinem Buch „Denkfallen und Programmierfehler“ entnommen. Der Regelkatalog ist der Feinabstimmung und Verbesserung zugänglich. Das geschieht im „Regelkreis des selbstkontrollierten Programmierens„ (Bild 3.1).

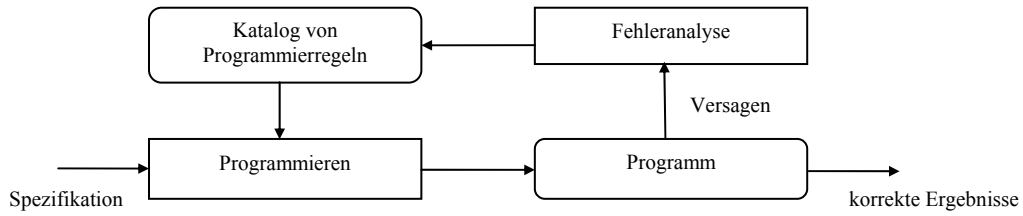


Bild 3.1 Der Regelkreis des selbstkontrollierten Programmierens

Viele der Regeln werden von der Informatikergemeinde mehrheitlich akzeptiert, so dass man sie zum Grundbestand eines guten Programmierstils rechnen kann. Wenn man sie befolgt, werden selbst große Probleme behandelbar. Die Entwurfsgrundsätze beschreiben allgemeine Vorgehensweisen und Zielsetzungen, die dafür sorgen sollen, dass unser Denkapparat auch bei voranschreitender Arbeit nicht unnötig überfordert wird.

Entwurfsgrundsätze

Die erste Gruppe von Regeln sind Entwurfsgrundsätze, die von der Gemeinde der Informatiker - zumindest im Bereich des Programmierens im Kleinen - akzeptiert werden.

- Auf *Lesbarkeit* achten

Erläuterung 1: Programme müssen lesbare Texte sein. Die Forderung nach Lesbarkeit impliziert die Forderungen nach größtmöglicher Einfachheit der Ablauf- und Datenstrukturen.

Erläuterung 2: Kommentare sollen dem Leser helfen, das Programm möglichst schnell zu verstehen. Sie sollen Gedächtnis und Denkapparat des Lesers entlasten. Das können sie nur, wenn sie knapp und treffsicher sind. Das bedeutet, dass der Denkaufwand für die Formulierung der Kommentare nicht hinter dem für die Codierung des Programms zurückstehen darf. Es hat sich bewährt, jedes Modul mit einer Kurzbeschreibung zu versehen, die den Zusammenhang zwischen Aufgabenstellung und Programmtext herstellt und auf verwendete Literatur hinweist. Außerdem ist es oft ratsam, die Bedeutung von Variablen zu erläutern. Gut platzierte Kommentare findet man direkt oberhalb des Kopfes einer Funktion und im Deklarationsteil eines Moduls. Beispiel:

```
//Hanoi.c
#include <stdio.h>

/*A, B, C: Quell-, Ziel und Hilfsturm*/
void move(char A, char B, char C, int n){
 if(n>1)  move(A, C, B, n-1);
 printf("%5d:%c ---> %c\n", n, A, B);
 if (n>1) move(C, B, A, n-1);
}

void main(){
 int n;
 printf("DIE TUERME VON HANOI\n");
 printf("? Anzahl der Scheiben = "); scanf("%d", &n);
 move('A', 'B', 'C', n);
}
```

Erläuterung 3: „Nutze die Routine und beachte Konventionen“ ist eine gute Regel für die Gestaltung von Bedienoberflächen (Grams, 2001, S. 110). Sie hilft auch bei der Verfassung lesbarer Programmtexte. Für Größenvergleiche gibt es beispielsweise das „<“- und das „>“-Zeichen. Die Ausdrücke „b>a“ und „a<b“ sind bedeutungsgleich. Die zweite Schreibweise nutzt die seit Schultagen in unserem Kopf vorhandene Vorstellung des Zahlenstrahls: Die kleineren Zahlen liegen links und die größeren rechts. Die *Abbildungstreue* von Schreibweise und Zahlenstrahl erleichtert das Lesen und Verstehen. Auf die „>“- und „>=“-Zeichen sollte man in Programmtexten also getrost verzichten.

- Vorzugsweise einfache *hierarchische Strukturen* erzeugen

Erläuterung: Das Programm wird aus hierarchisch angeordneten Funktionsblöcken (Moduln, Prozeduren) zusammengesetzt, so dass lauter überschaubare und voneinander weitgehend unabhängige Entwurfsaufgaben entstehen. In den höheren Hierarchiestufen befinden sich die problemnäheren, abstrakteren und übergeordneten Funktionsblöcke. Die niedrigeren Hierarchiestufen enthalten jeweils die erforderlichen Konkretisierungen.

- Nach der Methode der *schrittweisen Verfeinerung* (Stepwise Refinement) vorgehen

Erläuterung: Zunächst werden die Strukturen auf der höchsten (problemnahen und abstrakten) Ebene festgelegt. Dann konkretisiert man, von Stufe zu Stufe herabsteigend, die einzelnen Funktionsmoduln (Top-Down-Entwurf, Modularisierung). Änderungen sind stets auf der höchsten Hierarchiestufe durchzuführen, auf der sie sich auswirken (Auch hier: top-down vorgehen).

- Die Programmmoduln verbergen die Details ihrer Funktionsweise (*Information Hiding*)

Erläuterung: Das Information Hiding soll verhindern, dass schwer übersehbare Nebenwirkungen entstehen und dass Änderungen der Realisierung eines Moduls sich auf die den Modul benutzenden Teile auswirken.

- *Sparsamkeit* bei Schnittstellen- und Variablendeklarationen

Erläuterung: Es sind auf jeder Hierarchiestufe möglichst wenige Variablen zu deklarieren. Prozeduren arbeiten möglichst nicht mit globalen Variablen. Die Parameterlisten (Schnittstellen) sind äußerst knapp zu halten. Die Gültigkeitsbereiche von Variablen sind möglichst stark zu beschränken. Alle diese Maßnahmen dienen letztlich wieder der Lesbarkeit.

Überwindung der Kapazitätsbeschränkung

Eine Reihe weiterer Regeln trägt ebenfalls der Kapazitätsbeschränkung unseres Gedächtnisses Rechnung. Durch einfache Verwaltungs- und Gliederungsschemata und durch sorgsam platzierte Informationen wird Klarheit und Übersichtlichkeit erreicht.

- Verwendung einer möglichst geringen Zahl verschiedener *Strukturblocktypen*. Einschränkung auf
  - Sequenz (Reihung von Anweisungen)
  - Auswahl (IF- oder CASE-Anweisung)
  - Iteration (While-Schleife)

Verzicht auf freie Sprünge mittels GOTO.

Erläuterung: Die Beschränkung auf wenige Strukturblocktypen fördert die Lesbarkeit des Programms. Der Hauptgrund aus Sicht der strukturierten Programmierung ist aber, dass wir für diese Strukturen Beweisregeln kennen, beispielsweise den *Schleifensatz*.

- Auf aussagestarke Ausgabe achten

Erläuterung: Stets muss klar sein, was der Rechner gerade tut. Zu dem Zweck sind interne Zustände anzuzeigen und der Empfang von Eingaben in Form von Echos zu quittieren.

Gegen die negativen Folgen der Prägnanztendenz

Die Neigung zur Vereinfachung, die unpassende Verwendung von Faustregeln und Techniken der „abgestuften Qualität“ sind Ergebnis der Prägnanztendenz. Die folgenden Regeln gehen dagegen an.

- Fehlerkontrolle durchführen

Erläuterung 1: Die Durchführung von Simulationen (Berechnungsexperimenten mit dem Computer) erfordert besondere Aufmerksamkeit. Der Übergang vom mathematischen Modell (Differentialgleichungen und dergleichen) zum Algorithmus ist von systematischen Fehlern, beispielsweise aufgrund der Diskretisierung kontinuierlicher Größen, begleitet. Solche Fehler sind abzuschätzen und (am besten im Programm selbst) zu überwachen.

Erläuterung 2: Bei numerischen Programmen ist darüber hinaus stets die Maschinenarithmetik zu berücksichtigen. Insbesondere bei der Subtraktion etwa gleich großer Zahlen kann der relative Fehler des Ergebnisses recht groß werden. Die Hypothese, dass maschinenunabhängiges Programmieren numerischer Programme möglich ist, ist falsch! Das gilt jedenfalls für solche Programmiersprachen, die die Zahlenformate und numerischen Operationen nicht ausdrücklich festlegen.

- Nie reelle Zahlen auf Gleichheit oder Ungleichheit abfragen

Erläuterung: Wegen der Gleitpunktdarstellung können wir im Allgemeinen Zahlen nur bis auf einen Rundungsfehler genau darstellen. Wird eine Zahl auf verschiedenen mathematisch äquivalenten Wegen ermittelt, können die Rechenergebnisse differieren. Aber auch für diese Regel gibt es Ausnahmen: Wir werde später in gut begründeten Fällen Gleitpunktzahlen auf Gleichheit abfragen!

- Keine Wegwerf-Kontrollausdrucke verwenden

Erläuterung 1: Wenn etwas nicht richtig funktioniert, sollte man nicht gleich am Programm herummanipulieren. Zur Isolierung und Identifizierung von Fehlern kann man es zunächst einmal mit der Variation der Eingabe versuchen.

Erläuterung 2: Auf Kontrollausdrucke zum einmaligen Gebrauch ist möglichst zu verzichten. Druckbefehle, die zu Prüfzwecken in den Text eingestreut werden, sind sorgsam zu platzieren und nach Gebrauch in Kommentare zu verwandeln, so dass sie später erneut aktiviert werden können. Noch besser ist es, eine geeignete Teststrategie als wichtigen Programmbestandteil von Anfang an mit einzuplanen. Die Druckbefehle zur Unterstützung von Tests werden in Auswahlanweisungen untergebracht. So kann man sie über globale Steuerungsvariablen ein- und ausschalten.

- Faustregeln von Zeit zu Zeit überprüfen

Erläuterung: Vorgeblich effizienzsteigernden Tricks sollte man mit Argwohn begegnen. (Nicht glauben, wenn man nachsehen und prüfen kann.)

- Verwende Namen anstelle der Zahlendarstellungen. Verwende die standardisierten Bezeichner für die Systemkonstanten (Beispiele in der Programmiersprache C: `INT_MAX`, `FLT_EPSILON`, ...).

Erläuterung zur Entstehung dieser Regel: Diese Regel entstand im Rahmen der Lehrveranstaltung „Programmieren“ im Sommersemester 2002. In einem Programm fiel mir folgender Kontrollausdruck auf

```
scanf("%i", &n); if (n<-32768 || 32767<n) printf("Eingabefehler");
```

Ich: „So ist der Kontrollausdruck wirkungslos. Sie wollen nur Werte zulassen, die im darstellbaren Integer-Bereich liegen. Aber scanf sorgt durch die gewählte Formatierung bereits für die Einhaltung des Wertebereichs.“ Antwort: „Aber es funktioniert.“ Ich: „Aha, sie haben mit einem 32-Bit-Compiler übersetzt. Der Zahlenbereich ist also größer. Der Kontrollausdruck schränkt den Zahlenbereich unnötigerweise ein. Der Kontrollausdruck ist also nicht wirkungslos, sondern schädlich.“ Abgesehen davon, dass die Wertebereichseinschränkung hier keinen Sinn ergibt, sollte man grundsätzlich die vordefinierten Konstantenbezeichner für Minimal- und Maximalwerte verwenden. Das verbessert die Portabilität von Programmen.

Als es um die Einsortierung der Regel in das vorliegende Schema ging, wurde zunächst eine *Ursachenanalyse* durchgeführt. Eine Ursache liegt in der *Prägnanztendenz:* Strukturen tendieren dazu, sich zu verfestigen. Die Entscheidungen werden unter bestimmten Verhältnissen getroffen. Die Entscheidungen bestätigen und verfestigen die Strukturen in einer dem Bewusstsein verborgenen Rückkopplung. Der Programmierer ist außerdem Opfer der Denkfalle *Überbewertung bestätigender Informationen:* Dass die Sache zu funktionieren scheint, nämlich, dass scheinbar ungültige Eingaben abgewiesen werden, bestätigt den Programmierer darin, richtig zu liegen. Ein weiterreichendes Hinterfragen unterbleibt aufgrund der positiven Erfahrungen.

Die Prägnanztendenz wurde schließlich als die dominierende Ursache angenommen. Deshalb steht die Regel hier und nicht anderswo.

Gegen die negativen Folgen des Kausaldenkens

Es folgen ein paar Regeln, die unserem linearen Kausaldenken Rechnung tragen.

- *Redundanz* reduzieren

Erläuterung 1: Das Kopieren mehrfach verwendeter Programmteile schafft schwer zu überschauende Strukturen. Prozedurtechniken helfen, das zu vermeiden.

Erläuterung 2: Kommentare sind grundsätzlich redundant. Was die Funktion betrifft, können sie nur das ausdrücken, was der Programmtext ohnehin sagt. (Sie machen es dem Leser bestenfalls leichter.) Aufgrund von Programmänderungen kann es zu Abweichungen zwischen Programmfunktion und Kommentar kommen. Deshalb ist es nicht ratsam, ein Programm Zeile für Zeile zu kommentieren. Das gilt besonders für moderne Programmiersprachen, die die Erzeugung lesbarer Programmtexte unterstützen. Die Möglichkeiten, die man mit der freien Namensvergabe, der Erzeugung passender Datenstrukturen und der Einführung von Aufzählungstypen hat, sind weitgehend auszunutzen. Denn: die beste Dokumentation ist das gut strukturierte und lesbare Programm!

Erläuterung 3: Die Forderung nach Sparsamkeit der Kommentierung bedeutet keinesfalls, dass auf Kommentare verzichtet werden soll, und sie steht keinesfalls im Widerspruch zur Forderung nach Lesbarkeit. Gerade dann, wenn ein Algorithmus nicht ein naheliegendes Verfahren realisiert, sind Erklärungen nötig. Dann muss der Kommentar die Verbindung zwischen Aufgabenstellung und Programm herstellen. Man habe stets zwei Ziele im

Auge: Lesbarkeit des Textes und Entlastung des Denkapparats des Lesers. (Siehe auch die erste der Programmierregeln.)

- Überprüfung der *Korrektheit* von Funktionen und Operatoren

Erläuterung: Es ist möglichst die Methode des semi-algorithmischen Programmierens anzuwenden, denn diese Methode liefert neben dem Algorithmus auch einen *Korrektheitsnachweis* dafür. Außerdem stelle man sich immer wieder die folgenden Kontrollfragen: Ist die Funktion oder Operation für alle Werte des Definitionsbereichs richtig programmiert? Werden unzulässige Eingabewerte abgefangen?

- Ist ein Fehler gefunden: weitersuchen

Erläuterung: Es ist eine Erfahrungstatsache, dass in der Umgebung von Fehlern meist weitere zu finden sind.

- Benutze Entscheidungsbäume oder Entscheidungstabellen beim Aufstellen komplexer logischer Bedingungen

Überwindung der Überschätzung bestätigender Informationen

Die folgenden Regeln helfen, die Denkfalle „Überschätzung bestätigender Informationen", eine Konsequenz unserer Fähigkeit zur Induktion, zu überwinden:

- Planungsgrundsätze und Regeln für Tests beachten

Erläuterung: Die Arbeitsschritte Codierung, Fehlersuche, Fehlerbeseitigung sind klar zu trennen; die Testfälle werden in einem unabhängigen Arbeitsgang sorgfältig vorbereitet.

- Alternativen suchen

Erläuterung: Die Tatsache, dass die Sache augenscheinlich funktioniert, sollte noch kein Grund sein, sich mit der gefundenen Lösung zufrieden zu geben. Man sollte stets davon ausgehen, dass es noch bessere Lösungen gibt. Wichtig ist das Wegdenken von eingefahrenen Bahnen und von naheliegenden Ad-hoc-Lösungen. Den Blick weiten können

- die Literaturrecherche und
- das Programmieren auf der Basis von Korrektheitsbeweisen.

Irreführende Assoziationen zu vermeiden, ist Zweck der letzten Programmierregel:

- Benutze Namen, die die Variablen und Funktionen möglichst exakt bezeichnen

Aufgaben

3.1 *Sozialschwindler* (The Welfare Crook): Auf drei langen Magnetbändern sind Namen in alphabetischer Reihenfolge aufgelistet. Die erste Liste enthält die Angestellten von IBM Yorktown, die zweite, die Namen der Studenten der Columbia University, und die dritte die Namen der Sozialhilfeempfänger von New York. Die Listen können als endlos angenommen werden. Wenigstens ein Name befindet sich auf allen drei Listen. Schreiben Sie ein Programm, das die erste dieser Personen lokalisiert. Zur Vereinfachung nehmen wir an, dass die Namen auf drei Arrays stehen: `char *f[max],*g[max]` und `*h[max]` und dass als letzter Name ein Wächter (Sentinel) eingefügt ist, beispielsweise der „unmögliche" Name `ZZZ`.

3.2* *Halbkreis:* Schreiben Sie eine Funktion zur Deklaration `int Halbkreis(float a, b, c)`. Die Funktion soll genau dann den Wert 1 liefern, wenn die Zahlen a, b, c $\in$ [0, 1) auf einem Halbkreis liegen. Erläuterung: Das Intervall [0, 1) wird zu einem Kreis vom Umfang 1 „zusammengebogen".

4 Beweisgeleitetes Programmieren

Spezifikation

Wir spezifizieren die Funktion eines Programms oder eines Programmabschnitts *S* durch eine Vorbedingung pre(*S*) und eine Nachbedingung post(*S*). Das sind logische Ausdrücke (allgemeiner: Prädikate[3]), in denen die Programmvariablen vorkommen können, und die genau dann wahr sind, wenn die Wertebelegung der Variablen (der Zustand) unmittelbar vor bzw. unmittelbar nach der Programmausführung diese Bedingungen erfüllen. Ein Programm heißt (*vollständig*) *korrekt*, wenn es diese Spezifikation erfüllt, das heißt: Unter der Bedingung, dass pre(*S*) vor der Programmausführung gilt, endet das Programm in einem Zustand, in dem die Bedingung post(*S*) erfüllt ist.

Ein Programm heißt *partiell korrekt*, wenn unter der Bedingung, dass pre(*S*) vor Programmausführung gilt, das Programm entweder kein Resultat liefert, oder aber ein Resultat liefert, das die Nachbedingung post(*S*) erfüllt.

Unter einem Programmbeweis versteht man den Nachweis der Korrektheit mit formalen oder mathematischen Methoden. Eine dieser Methoden ist die symbolische Programmausführung.

Beweisgeleitetes Programmieren lässt sich am besten durch folgendes Prinzip charakterisieren: „Ein Programm und sein Beweis sollten Hand in Hand entwickelt werden, wobei der Beweis den Weg weist“ (Gries, 1981, S.). Ferner meint Gries, dass Programmieren eine zielorientierte Tätigkeit sei: Ausgehend von der Spezifikation wird die Aufgabe in Teilaufgaben unterteilt, für die wiederum Spezifikationen abgeleitet werden. Schließlich werden die Teile realisiert.

Das beweisgeleitete Programmieren ist also gekennzeichnet durch das „Denken vom Resultat her“. Auf diese Art der Programmierung passt die Bezeichnung „semi-algorithmisch“: Ist die Spezifikation einmal gegeben, dann hat der Konstrukteur auf jeder Stufe des Konstruktionsprozesses zwar gewisse Entscheidungsmöglichkeiten (deshalb „semi“), aber alle führen - bei konsequenter Anwendung der Methode - nahezu zwangsläufig zu richtigen Lösungen (deshalb „algorithmisch“).

Dieses Entwurfs- bzw. Konstruktionsverfahren lässt sich durch weitere Schlagworte charakterisieren: Programmieren auf der Basis von Korrektheitsbeweisen, diskursive Methode, schrittweise Verfeinerung (stepwise refinement), Top-Down-Design.

Die Methode ist Gegenstand einer ganzen Reihe von Lehrbüchern: Alagic/Arbib (1978), Gries (1981), Baber (1987), Linger/Mills/Witt (1979).

Symbolische Programmausführung

Der Korrektheitsnachweis durch *symbolische Programmausführung* geschieht in folgenden Teilschritten:

1. Anfangsbelegung und Übersetzung: Anstelle der konkreten Zahlenwerte für die Variablen belegen wir die Programmvariablen mit Variablen im mathematischen Sinn - also mit Va-

[3] Ein Prädikat ist - ebenso wie ein boolescher Ausdruck - eine Funktion, die nur die Werte falsch (0) und wahr (1) annehmen kann. Allerdings sind die Variablen und die Operatoren nicht mehr allein auf die booleschen Variablen und Operatoren beschränkt. Beispiele für Prädikate sind die booleschen Ausdrücke in den üblichen Programmiersprachen. Aber es sind auch Formulierungen zulässige wie "Für alle x gilt ..." (Allquantor) und "Es gibt ein x, so dass ..." (Existenzquantor).

riablen, deren Werte zwar beliebig gewählt aber fest sind[4]. Die Vorbedingung des Programmabschnitts wird in die Sprache der Mathematik übersetzt, indem die Programmvariablen durch ihre momentanen Werte, also die jeweiligen mathematischen Variablen ersetzt werden. Programmvariablen schreiben wir mit Kleinbuchstaben. Die mathematischen Variablen werden groß und kursiv geschrieben, um sie von den Programmvariablen möglichst gut zu unterscheiden.

2. Symbolische Programmausführung: Dann führen wir die Anweisungen des zu untersuchenden Programmabschnitts Schritt für Schritt mit diesen mathematischen Variablen - also symbolisch - aus. Als Endzustand ergibt sich eine bestimmte Wertebelegung der Programmvariablen.

3. Mathematische Umformungen: Nun stehen auf den Programmvariablen Ausdrücke, in denen nur noch diese mathematischen Variablen vorkommen. Der Gültigkeitsnachweis für irgendwelche Aussagen findet ausschließlich im Bereich der Mathematik statt.

4. Rückübersetzung: Schließlich werden die mathematischen Variablen wieder aus den mathematischen Beziehungen eliminiert, indem die Programmvariablen anstelle ihrer nun gültigen Werte eingesetzt werden. Dadurch ergeben sich die am Ende gültigen Zusicherungen. Der Programmbeweis ist gelungen, wenn diese Zusicherungen die Nachbedingung implizieren.

Für jeden möglichen Pfad durch einen zu beweisenden Programmabschnitt ist der Korrektheitsbeweis zu führen. Alle möglichen Pfade durch einen Programmabschnitt lassen sich mit dem Kontrollflussgraphen ermitteln.

Der *Kontrollflussgraph* ist ein Diagramm der Ablaufstruktur, in dem die Anweisungen oder Anweisungssequenzen durch Knoten repräsentiert werden. Die Pfeile dienen zur Darstellung von Übergängen zwischen Programmteilen. In reduzierter Darstellung können Sequenzen von Knoten auch weggelassen werden. Dann sind die Anweisungen dem entsprechenden Pfeil zugeordnet.

Beispiel 1: Vor- und Nachbedingung des C-Programmabschnitts "i++; s+=i;" seien gegeben durch den jeweils gleichen Ausdruck s=i(i+1)/2. Also:

```
pre("i++; s+=i;") =(s=i(i+1)/2) und
post("i++; s+=i;")=(s=i(i+1)/2).
```

Dass der Programmabschnitt korrekt ist, wird jetzt mit der Methode der symbolischen Ausführung bewiesen.

1. Die Anfangsbelegung ist gegeben durch (i, s) = (*I*, *S*). Die Vorbedingung ist gleich $S = I(I+1)/2$.

2. Die symbolische Programmausführung liefert die folgenden Zustände (Wertebelegungen): Nach der Anweisung „i++;“ ist der Zustand gegeben durch (i, s) = (*I*+1, *S*) und nach der Anweisung „s+=i“ ist der Zustand gegeben durch (i, s) = (*I*+1, *S*+*I*+1).

3. Wir lösen die Beziehungen zwischen den mathematischen Variablen und den Programmvariablen nach den mathematischen Variablen auf: I = i-1, S = s-i.

---

[4] In der Mathematik sind Variablen *Namen* für Werte. In der Informatik sind Variablen *Behälter* für Werte.

4. Nun ersetzen wir in der Formel $S = I(I+1)/2$ die mathematischen Variablen wieder durch die Programmvariablen und erhalten s-i=(i-1)i/2. Nach einfachen Umformungen ergibt sich daraus die Nachbedingung s=i(i+1)/2. Damit ist der Korrektheitsbeweis erbracht.

Beispiel 2: Der Programmabschnitt *P* sei gegeben durch

$P =$ „`if (a<b) min=a; else min=b;`"

Den zugehörigen Kontrollflussgraph zeigt Bild 4.1. Er soll für beliebige Wertebelegungen für min das Minimum der Werte a und b liefern. Genauer: Das Programm soll bezüglich der folgenden Vor- und Nachbedingungen korrekt sein:

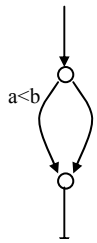


Bild 4.1 Kontrollflussgraph

`pre(`P`): a =` A`; b=`B

`post(`P`): a =` A`; b=`B`;` $(A<B)\wedge(\min=A) \vee \neg(A<B)\wedge(\min=B)$

Hierin haben Doppelpunkt und Semikolon die Bedeutung der logischen Gleichheit bzw. der UND-Verknüpfung. Von den normalen Verknüpfungen unterscheidet sie nur der Vorrang: Beide Verknüpfungen haben niedrigere Priorität als alle sonstigen logischen Verknüpfungen. Die niedrigste Priorität hat der Doppelpunkt. Durch diese Festlegungen lassen sich einige verwirrende Klammern einsparen.

Anfangs ist die Wertebelegung gegeben durch (a, b, min) = (*A*, *B*, ?). Das Durchlaufen des linken Pfades führt auf die Bedingung ($A<B$) und die Wertebelegung (a, b, min)=(*A*, *B*, *A*). Das läuft auf die Gültigkeit des Ausdrucks $(A<B)\wedge(a = A)\wedge(b=B)\wedge(\min = A)$ hinaus. Und dieser Ausdruck impliziert die Nachbedingung. Bei Durchlaufen des anderen Pfades ergibt sich die Gültigkeit von $\neg(A<B)\wedge(a = A)\wedge(b=B)\wedge(\min = B)$. Auch dieser Ausdruck impliziert die Nachbedingung. Da nach Durchlaufen eines jeden der möglichen Pfade ein Ergebnis herauskommt, das die Nachbedingung gültig macht, ist das Programm korrekt.

Aufgaben

4.1 *Potenz.* Zeigen Sie, dass die Anweisung „`if (n%2) {n--; y*=x;} else {n/=2; x*=x;}`" bezüglich der Vorbedingung $y*x^n = A$ und derselben Nachbedingung korrekt ist. Mit *A* wird eine geeignete (vorerst noch unbekannte) reelle Zahl bezeichnet. Schreiben Sie ein effizientes Programm zur Berechnung der *N*-ten Potenz einer reellen Zahl *X* unter Ausnutzung dieser Anweisung. *Hinweis:* y ist anfangs gleich 1 und schließlich gleich dem gesuchten Ergebnis. Weisen Sie die Korrektheit des Programms nach: Zeigen Sie, dass es ein Ergebnis liefert und dass das gelieferte Ergebnis auch richtig ist.

4.2 *Effizienz der Potenzberechnung.* Schätzen Sie die Effizienz des Potenz-Programms ab: Geben Sie eine möglichst genaue Abschätzung der maximal notwendigen Schleifendurchläufe in Abhängigkeit von *N* an. Schreiben Sie ein Testprogramm, das für verschiedene Werte von *X* und *N* das gewünschte Resultat zusammen mit der tatsächlichen Anzahl von Schleifendurchläufen sowie den Schätzwert dafür ausgibt. Alle Algorithmen sollen ausschließlich ganzzahlige Variablen verwenden.

Zusatz: Führen Sie die Korrektheitsnachweise für sämtliche Programmteile durch, sobald Sie den Schleifensatz des folgenden Abschnitts kennen gelernt haben. Im Rahmen der Beweisführung wird die div-mod-Identität benötigt[5]: $z = (z \text{ div } m)m + z \bmod m$.

---

[5] Quelle: „Ein Beitrag zum div-mod-Problem" von Timm Grams in der Overflow-Kolumne von Jürg Nievergelt („Über das div-mod-Problem und die Normierung ganzzahliger Arithmetik sowie ein Rückblick auf Zahlenkreuze") im Informatik-Spektrum Band 14 (Dezember 1991), S. 351-354

5 Algorithmenentwurf mittels Invariante

Der Schleifensatz

Iterationsschleifen sind die Hauptstrukturelemente vieler Algorithmen. Eine formale Behandlung solcher Schleifen liegt nahe, weil gerade hier Entwurfsfehler besonders häufig zu beobachten sind. Im folgenden wird eine einfache Version des Schleifensatzes dargestellt. In dieser Form reicht er für While-Schleifen, wie sie beispielsweise in PASCAL oder MODULA-2 möglich sind und wie sie sich auch in anderen prozeduralen Sprachen wie FORTRAN, C bzw. C++ und ALGOL realisieren lassen.

In Pascal schreibt man „WHILE *B* DO *S*“ und in C „while (*B*) *S*“. Hierin bezeichnet *B* die *Schleifenbedingung* und S die *Schleifenanweisung*, manchmal auch *Schleifenkörper* genannt.

Für die While-Schleife wird eine *Endlichkeitsbedingung* definiert: Die Schleife ist für den Zustand σ endlich, wenn es ein n, $0 \le n < \infty$, gibt, so dass

1. $\neg B(S^n(\sigma))$ gilt und außerdem
2. $B(S^i(\sigma))$ für alle $0 \le i < n$

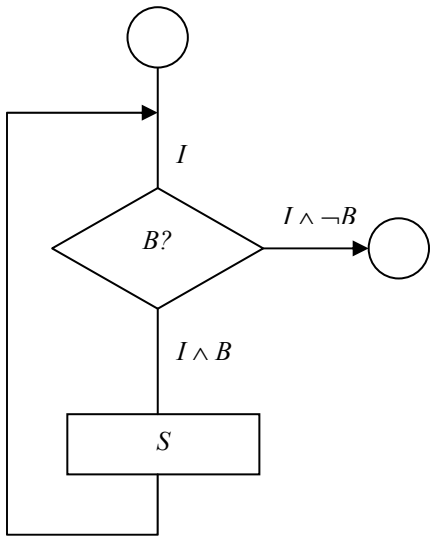


Bild 5.1 Der Schleifensatz

Bei der Endlichkeitsbedingung handelt es sich also um ein Prädikat, das von *B* und *S* abhängt und das genau dann wahr ist, wenn die wiederholte Anwendung von *S* (eventuell auch nullmalig) schließlich auf einen Zustand führt, in dem *B* falsch ist und wenn bis dahin nur Zustände auftreten, für die sowohl *B* als auch *S* definiert sind und für die *B* wahr ist.

Falls für einen Zustand σ die Endlichkeitsbedingung erfüllt ist, liegt auch die Zahl *n* fest. Man sagt dann, dass die Schleife nach *n* Iterationen endet.

Die Endlichkeitsbedingung legt den Definitionsbereich der While-Schleife fest. Die Endlichkeitsbedingung wird in folgenden Fällen nicht erfüllt:

1. $B(S^n(\sigma))$ ist für alle *n* wahr und die Berechnung bricht nicht ab (Endlosschleife)
2. Es entsteht im Verlaufe der Iterationen ein Zustand, für den *B* oder *S* nicht definiert ist

Für das algorithmenorientierte Vorgehen sind Prädikate von besonderem Interesse, die vor und nach jedem Iterationsschritt wahr sind. Ein solches Prädikat heißt *Invariante*. Wenn es gelingt, Prädikate *I* und *B* zu finden, so dass das Prädikat

$$I \wedge \neg B$$

das gewünschte Resultat *R* - die Nachbedingung - impliziert ($I \wedge \neg B \le R$); wenn sich ferner eine Anweisung oder Rechenvorschrift *S* finden lässt, die *I* invariant lässt, solange *B* gilt; wenn ferner die wiederholte Anwendung von *S* (unter der Vorbedingung *I*) schließlich auf einen Zustand führt, in dem *B* nicht mehr gilt, dann liefert die Schleife eine Lösung des Problems, sofern man sicherstellt, dass zu Beginn die Invariante *I* wahr ist.

Diese Aussage lässt sich sehr prägnant in der Form des *Schleifensatzes* ausdrücken: Falls *I* die Endlichkeit der While-Schleife impliziert und wenn *S* bezüglich der Spezifikation pre(*S*) = $I \wedge B$ und post(*S*) = *I* korrekt ist, dann ist die Schleife bezüglich der Spezifikation pre(„while (*B*) *S*“) = *I* und post(„while (*B*) *S*“)= $I \wedge \neg B$ vollständig korrekt. Bild 5.1 veranschaulicht den Schleifensatz anhand eines Programmablaufplans.

Programmierstudie: Ganzzahlige Quadratwurzel

Aufgabe: Ein Programm *P* soll den maximalen Wert bestimmen, dessen Quadrat nicht größer als die gegebene Zahl x ist (y soll schließlich also gleich der ganzzahligen Quadratwurzel von x sein). Alle Variablen sind ganzzahlig.

Lösung: Aus der Aufgabenstellung ergibt sich folgende *Spezifikation* für das Programm *P*

$pre(P): 0 \le x$

$post(P): (0 \le y) \wedge (y^2 \le x < (y+1)^2).$

Das Programm *P* wird als While-Schleife mit Initialisierungsteil angesetzt. Die *Schleifeninvariante I* wird folgendermaßen definiert

$I: (0 \le y) \wedge (y^2 \le x)$

Die Schleife soll abbrechen, sobald $x<(y+1)^2$ gilt (*Abbruchbedingung*). Die *Schleifenbedingung B* wählt man naheliegenderweise gleich der Negation dieser Bedingung:

$B: \neg(x<(y+1)^2)$

B ist äquivalent zu $(y+1)^2 \le x$.

Das folgende Programmstück ist - abgesehen von der Syntax der Schleifenbedingung - eine naheliegende und korrekte Realisierung von *P:*

```
y=0;
while ((y+1)^2 ≤ x) y++;
```

Korrektheitsbeweis: Der Initialisierungsteil y= 0 macht die Invariante wahr. Wir können also davon ausgehen, dass bei Eintritt in die Schleifenanweisung y++ die $I \wedge B$ gilt. Nun wird mittels symbolischer Ausführung nachgewiesen, dass *I* Invariante der Schleifenanweisung ist. Zu zeigen ist also, dass y++ korrekt ist bezüglich der Spezifikation

$pre(„y++;"): (0 \le y); (y^2 \le x); (y+1)^2 \le x$

$post(„y++;"): (0 \le y); (y^2 \le x)$

Die Vorbedingung lässt sich noch vereinfachen, weil ja $(y+1)^2 \le x$ die Bedingung $y^2 \le x$ impliziert:

$pre(„y++;"): 0 \le y; (y+1)^2 \le x$

Der Zustand bei Eintritt in die Schleifenanweisung sei gegeben durch (x, y) = (*X*, *Y*). Da die Vorbedingung voraussetzungsgemäß erfüllt sein soll, haben wir für die mathematischen Variablen die Beziehungen

$0 \le Y$ und $(Y+1)^2 \le X$.

Nach Abarbeitung der Schleife ist der Zustand (x, y) = (*X*, *Y*+1) erreicht. Wir machen die Rückübersetzung und ersetzen in den Beziehungen *X* durch *x* sowie *Y* durch *y*-1. Es ergeben sich die Relationen

$0 \le y-1$ und $y^2 \le x$.

Jedenfalls gilt dann auch $0 \le y$ und $y^2 \le x$ und das ist die Invariante *I*. Damit ist die Invariantenteneigenschaft nachgewiesen.

Übung: Weisen Sie nach, dass die Invariante die Endlichkeit der Schleife impliziert.

Nach Beendigung der Schleife gilt $I \wedge \neg B$ und das ist gleich $(0 \le y) \wedge (y^2 \le x) \wedge (x<(y+1)^2)$. Dieser Ausdruck ist gleich post(*P*). Damit ist die Korrektheit des Programms *P* bewiesen.

Man könnte nun die Schleifenbedingung in korrekter Syntax aufschreiben und hätte ein funktionierendes und korrektes Programm gewonnen.

Wir suchen aber nach einer noch einfacheren Lösung und Verwenden den Schleifensatz zur Optimierung des Programms.

Ausgangspunkt für die Optimierung ist der Gedanke das Herüberretten von Rechenergebnissen von einem Schleifendurchlauf in den darauf folgenden: Im Schleifendurchlauf wird der Wert $(y+1)^2$ berechnet, und dieses Ergebnis nützt vielleicht beim nächsten, wo $(y+1+1)^2$ zu berechnen ist. Also führen wir eine neue Variabel w ein, die folgendermaßen definiert ist: $w=(y+1)^2$. Diese Beziehung muss eine Invariante der Schleife sein. Seien nun y und w die unmittelbar vor dem Schleifendurchlauf erreichten Werte. Dann ergibt sich innerhalb der Schleife für y der neue Wert $y' = y+1$ und der neue Wert von w ist gegeben durch $w' = (y'+1)^2 = y'^2+2y'+1 = (y+1)^2+2y'+1 = w+2y'+1$.

Daraus folgt, dass die Sequenz „y++; w+= 2*y+1“ den Ausdruck $w=(y+1)^2$ invariant lässt. Der Algorithmus nimmt damit folgende Form an

```
y=0; w=1;
while (w <= x) {y++; w+=2*y+1;}
```

Durch den Erfolg mutig geworden, wollen wir versuchen, durch Einführung einer weiteren Variablen z den Aufwand weiter zu reduzieren. Wir definieren z=2*y+1 und müssen nun dafür sorgen, dass diese Gleichung eine Schleifeninvariante wird. Damit ergibt sich schließlich dieses *Quadratwurzelprogramm*:

```
y=0; w=1; z=1;
while (w<=x) {y++; z+=2; w+=z;}
```

Programmierstudie: Wortsuche

Die Wortsuch-Aufgabe habe ich in einem Aufsatz von Niklaus Wirth gefunden (Spektr. d. Wiss. 11/1984, S. 46-58). Wirths Lösungsvorschlag zeigt: Auch die Meister der Programmierung machen manchmal Fehler. Diese Entdeckung bereicherte meine Sammlung der „Denkfallen beim Programmieren“ um ein interessantes Beispiel.

Robert Baber schlug dann eine besonders einfache Lösung vor, die er aus einer *Invarianten* heraus entwickelt hat. Eine vollständige und formale Behandlung der Programmierstudie ist in meinen „Denkfallen und Programmierfehler“ (1990) zu finden. Auch Baber hat in seinem „Zauberlehrling“ (1990) diese Lösung dargestellt. Hier die

Aufgabe: Es ist ein Text t[M] nach einem Wort w[N] zu durchsuchen. Das Ergebnis soll ein Verweis (Index) auf das erste Vorkommen des Wortes sein.

Lösungsidee: Um festzustellen, ob das Wort auf Position i des Textes steht, vergleicht man w[0] mit t[i], w[1] mit t[i+1] usw. Die Prüfung wird so lange fortgesetzt, wie die Buchstaben übereinstimmen. Sowie eine Abweichung festgestellt wird, geht man zur nächsten Position und fängt von vorne an. Das Ganze macht man für i = 0, 1, 2, ... solange, bis einmal Übereinstimmung für alle Buchstaben des Wortes w herrscht, oder bis man am Ende des Textes angelangt ist.

Invariante: Diese Lösungsidee lässt sich formal mit Hilfe einer Invarianten präzise fassen. Zunächst bezeichnen wir mit dem Prädikat *P*(i, k) den Sachverhalt, dass ab dem Index i des Textes die Buchstaben mit den ersten k-ten Buchstaben des Wortes (also bis einschließlich w[k-1]) übereinstimmen: $P(i, k) = (w[0]=t[i] \wedge w[1]=t[i+1] \wedge ... \wedge w[k-1]=t[i+k-1])$.

Das Prädikat P ist Ausgangsbasis zur Formulierung der Bedingungen für das Programm. Insbesondere lässt sich der Sachverhalt, dass das gesuchte Wort tatsächlich auf Position i zu finden ist, durch $P(i, N)$ ausdrücken.

Der Text t wird systematisch von vorn nach hinten nach dem Wort w abgesucht: i = 0, 1, 2, ... Der Stand der Bearbeitung lässt sich so ausdrücken: An keiner Position vor j kommt das Wort vor. Ab Position j stimmen die Buchstaben mit den ersten k Buchstaben des Wortes w überein. Formal lässt sich das mit dem folgenden Prädikat sagen: $((0 \le i < j) \le \neg P(i, N)) \wedge P(j, k)$. Hierin ist das (kursiv geschriebene) i eine Variable, die jeden belieben ganzzahligen Wert annehmen kann. Dieses Prädikat ist die Invariante I des gesuchten Algorithmus. Also

$I = ((0 \le i < j) \le \neg P(i, N)) \wedge P(j, k)$.

Grobentwurf: Die Anforderungen an den Algorithmus lassen sich jetzt präzisieren, wenn wir als Grundstruktur eine Schleife wählen:

```
Init;
while (k<N&&j<=M-N ) {erhöhe (j, k) und erhalte I}
```

Feinentwurf der Initialisierung: Die passende Initialisierung ist schnell gefunden: Für (j, k) = (0, 0) ist *I* wahr. Folglich reicht zur Initialisierung die Sequenz j:= 0; k:= 0.

Feinentwurf des Schleifenkörpers: Was heißt: erhöhe (j, k)? Offensichtlich ist zur Erreichung des Schleifenendes danach zu trachten, dass j und k möglichst wachsen. Falls w[k]=t[j+k] gilt, kann man k erhöhen und die Invariante bleibt wahr. Andernfalls muss man versuchen, j zu erhöhen. Eine Erhöhung von j setzt die Erfüllung mehrerer Bedingungen voraus. Eine davon ist, dass P(j, k) wahr bleiben muss. Das lässt sich durch Zurücksetzen von k auf den Wert null erreichen. Diese Überlegungen legen nahe, die Wertepaare (j, k) lexikographisch zu ordnen: Es sei (j, k) < (j', k') genau dann, wenn j<j' oder wenn j=j' und k<k'. Im Sinne dieser Ordnungsrelation wollen wir „erhöhe (j, k)“ auffassen.

Für den Programmabschnitt „erhöhe (j, k) und erhalte I“ bietet sich eine Auswahlanweisung an. Danach sieht das endgültige *Wortsuche-Programm* so aus:

```
j=0; k=0;
while (k<N&&j<=M-N) if (w[k]==t[j+k]) k++; else {j++; k=0;}
```

Aufgaben

5.1 Beweisen Sie mit der Methode der symbolischen Programmausführung, dass $w=(y+1)^2$ und z=2*y+1 tatsächlich Schleifeninvarianten des Quadratwurzelprogramms sind.

(CACM-Algorithmus 246).

5.2 Zeigen Sie, dass der Schleifenkörper des Wortsuche-Programms die Invariante tatsächlich erhält.

5.3 Nach Durchlaufen der Schleife gilt die Invariante und die negierte Schleifenbedingung. Interpretieren Sie das Ergebnis für das Wortsuche-Programm.

5.4 Es ist ein Programm zu schreiben, das die reelle Zahl b auf den Maximalwert der Komponenten des Feldes `int x[n]` setzt. Beginnen Sie mit der Spezifikation. Machen Sie dann Grobentwurf und Feinentwurf und beweisen Sie die Korrektheit ihres Algorithmus. *Hinweis:* Führen Sie das Prädikat kg(i) ein mit der Bedeutung, dass a[j] ≤b für alle j mit 0≤j<i gilt und dass es ein k gibt, so dass 0≤k<i und x[k]=b.

5.5 Es ist ein Programm zu schreiben, das für die boolesche Variable b genau dann den Werte true liefert, wenn alle Komponenten des Feldes `int a[n]` gleich sind. Vorgehen wie oben.

5.6 *Nationalflagge:* Das Zeichenarray `f` (Deklaration: `char f[n];`) ist mit den Zeichen `s`, `r` und `g` gefüllt. Das heißt: auf jeder Position des Arrays ist ein `s`, ein `r` oder ein `g` eingetragen worden, und zwar rein zufällig mit der jeweiligen Wahrscheinlichkeit von 1/3. Schreiben Sie ein Programm, das die Zeichen in der Reihenfolge `ss...srr...rgg... g` anordnet. Ermitteln Sie den mittleren Rechenaufwand ihres Programms: Wählen Sie n=1000, füllen Sie das Zeichenarray rein zufällig mit den Buchstaben `s`, `r` und `g`. Ermitteln Sie die Häufigkeit der Operationen, mit denen ihr Programm auf das Array zugreift (lesend und schreibend). Führen Sie dieses Experiment 1000 mal durch und ermitteln Sie die mittlere und die maximale Häufigkeit dieser Operationen.

5.7* *LexikalischePermutationen:* Die Ziffern einer n-stelligen Dezimalzahl sind so zu permutieren, dass die nächstgrößere auf diese Weise erhältliche entsteht (Gries, 1981, S. 178; CACM-Algorithmus 323)

5.8* *ZufaelligePermutationen:* Die Ziffern einer n-stelligen Dezimalzahl sind durch einen Zufallsprozess so umzustellen, dass sich für jede mögliche Permutationen dieselbe Wahrscheinlichkeit ergibt (CACM-Algorithmen 235/362).

5.9* *GrayCode:* Gesucht ist ein Programm, das sämtliche 2^n n-stelligen Bitkombinationen so anordnet, dass sich zwei aufeinanderfolgende Bitkombinationen in nur einer Stelle unterscheiden

6 Der Kreisalgorithmus von Bresenham

Programmierstudie

Die Idee zu dieser Lektion ist von Wirth (1990). Im Buch von Wolf-Dietrich Fellner (1988) findet man weitere Kreisalgorithmen neben dem von Bresenham.

Der *Kreisalgorithmus von Bresenham* ist verblüffend einfach, enorm effizient - und auf den ersten Blick nicht zu durchschauen. Man fragt sich: Tut er überhaupt das, was er soll, nämlich näherungsweise Kreise in ein Raster zeichnen? Wieso funktioniert die Sache? Welche Lösungsidee steckt dahinter? Wie kommt man auf solch eine Lösung?

Zur Klärung der ersten Frage probiert man den Algorithmus einfach aus. Das ist zwar kein Beweis, schafft aber Vertrauen. Volle Klarheit und Antworten auf die übrigen Fragen erhält man auf konstruktivem Weg: Man versucht selbst, den Algorithmus zu finden und - nachdem man eine erste Lösung hat - ihn Schritt für Schritt zu verbessern.

Aber fangen wir mit der präzisen *Aufgabenstellung* an:

Gesucht ist ein Algorithmus, der in einem zweidimensionales Raster näherungsweise einen Achtelkreis mit dem Radius r um den Nullpunkt zeichnet. Der Abstand der Rasterpunkte in horizontaler und vertikaler Richtung ist jeweils gleich eins. Der Achtelkreis ist durch die Bedingung $0 \le x \le y$ für die Kreispunkte (x, y) festgelegt (zweiter Oktant). Den vollen Kreis erhält man damit leicht aufgrund der Symmetrie.

Den Einstieg in den Entwurf liefert folgender naheliegende Gedanke: In einer Schleife wird - ausgehend vom Wert 0 - die x-Koordinate schrittweise vergrößert. Innerhalb der Schleife wird y soweit angepasst, dass der Punkt möglichst nahe an den idealen Kreis herankommt. Die Kreisgleichung ist $x^2 + y^2 = r^2$. Da die Rasterpunkte im Allgemeinen den Kreis nicht genau treffen, wird die Größe

$$x^2 + y^2 - r^2 \qquad \text{(A1)}$$

nur im Idealfall gleich null sein. Liegt der Punkt (x, y) auf einem größeren Kreis, dann ist der durch (A1) gegebene Wert positiv; ist der Kreis kleiner, dann wird der Wert negativ.

Eingehendere Untersuchung verdient der Fall, dass der Punkt (x, y) gerade außerhalb und der Punkt (x, y-1) innerhalb des Kreises liegt: Welcher von beiden Punkten ist zu nehmen? Für den zweiten Punkt ergibt sich anstelle von (A1) der Wert

$$x^2 + (y-1)^2 - r^2 \qquad \text{(A2)}$$

Im betrachteten Grenzfall ist dieser Wert negativ. Die Summe der Werte (A1) und (A2) ist

$$2x^2 + 2y^2 - 2y + 1 - 2r^2 \qquad \text{(A3)}$$

Dieser Wert ist positiv, wenn beide Punkte außerhalb des gesuchten Kreises liegen und negativ, wenn innerhalb. Falls einer innerhalb und einer außerhalb des Kreises liegt, wird der Absolutwert minimal. Er ist positiv, wenn der innere - nämlich (x, y-1) - näher an der Kreisperipherie liegt, und negativ, wenn der äußere - nämlich (x, y) - näher dran ist.

Die letzte Bemerkung ist noch zu begründen. Da nicht die Radien sondern deren Quadrate verglichen werden, ist die Aussage nur im Sinne der folgenden Näherung richtig: Sei $r+\delta_1$ der Radius des äußeren und $r-\delta_2$ derjenige des inneren Kreises. Dann entspricht (A1) offenbar dem Wert $(r+\delta_1)^2 - r^2$, und der ist gleich $2r\delta_1 + \delta_1^2$. Da meist $|\delta_1|$ wesentlich kleiner als r ist, nimmt (A1) ungefähr den Wert $2r\delta_1$ und (A2) dementsprechend den Näherungswert $-2r\delta_2$ an. Der Näherungswert des Ausdrucks (A3) ist die Summe der beiden, also $2r(\delta_1-\delta_2)$. Ob der Ausdruck (A3) positiv oder negativ ist, hängt also direkt davon ab, ob δ_1 oder δ_2 der größere der beiden Werte ist.

Daraus folgt das *Optimalitätskriterium* für den Wert y: Bei festgehaltenem Wert x ist das maximale y zu wählen, für das der Ausdruck (A3) negativ ist.

Aus diesen Vorüberlegungen ist klar, wie der Algorithmus vorzugehen hat: Falls der Wert (A3) nichtnegativ ist, wird y solange verringert, bis der Wert negativ wird. Genau in diesem Moment ist der günstigste Punkt (x, y) erreicht.

Zu Beginn, also wenn x = 0 ist, wird y = r gesetzt. Damit enthält man den *Grobentwurf:*

```
x= 0; y= r;
while (x≤y) {
  mark(x, y);
  x++;
  while (0 ≤ 2x² + 2y² - 2y + 1 - 2r²) y--;
}
```

In diesem Entwurf sind die Ausdrücke noch nicht C-like, sondern im Pseudocode geschrieben, und die Schleifenbedingung der inneren While-Schleife ist überdies ziemlich kompliziert. Mit Hilfe des Schleifensatzes wird nun versucht, die Sache weiter zu vereinfachen und vor allem effizienter zu machen.

Feinentwurf: Wir setzen

$$h = 2x^2 + 2y^2 - 2y + 1 - 2r^2 \qquad (I)$$

und führen diese Beziehung als Invariante für die beiden Schleifen ein. Es fragt sich nun, wie man die Gültigkeit der Invarianten (*I*) aufrechterhält, wenn sich x und y den Anweisungen des Programms entsprechend verändern.

Zunächst nehmen wir den Fall, dass sich x um 1 erhöht: x' = x + 1. Das neue h ist gegeben durch

$$\begin{aligned} h' &= 2(x+1)^2 + 2y^2 - 2y + 1 - 2r^2 \\ &= 2x^2 + 4x + 2 + 2y^2 - 2y + 1 - 2r^2 \\ &= h + 4x + 2 \end{aligned}$$

Die Änderung von h stellt man also der Zuweisung x++ voran. Die Veränderung von y ergibt als neuen Wert y' = y-1 und h ist dementsprechend folgendermaßen zu verändern

$$\begin{aligned} h' &= 2x^2 + 2(y-1)^2 - 2(y-1) + 1 - 2r^2 \\ &= 2x^2 + 2y^2 - 4y + 2 - 2y + 2 + 1 - 2r^2 \\ &= h - 4y + 4 \\ &= h - 4(y-1) \\ &= h - 4y' \end{aligned}$$

Hier empfiehlt es sich, bei der Veränderung von h auf den neuen Wert y' von y zuzugreifen, also die Veränderung von h erst nach der Änderung von y durchzuführen. Die Zuweisungssequenz „y--; h-= 4*y“ erhält die Invariante.

Damit - und mit einer geeigneten Ergänzung des Initialisierungsteils - erhält man das folgende *Kreisalgorithmus-Programm:*

```
x=0; y=r;
h= 1-2*r;
while (x<=y) {
  mark(x, y);
  h+= 4*x+2;
  x++;
  while (0<=h){y--; h-=4*y;}
}
```

Feinabstimmung: Beim Kreisalgorithmus in der oben angegebenen Variante kann man die innere While-Schleife durch eine einfache If-Anweisung ersetzen, da mehr als ein Anpassungsschritt der y-Komponente aus geometrischen Gründen nicht vorkommt. Falls ein Anpassungsschritt geschieht, wird die Variable h innerhalb der äußeren Schleife zweimal im Wert geändert. Das lässt sich vermeiden. Da innerhalb der Schleife die x-Komponente jedenfalls verändert wird, kann man die Schleifeninvariante so formulieren, dass die Variable h bis zur Abfrage 0<=h nicht geändert werden muss; nur für diese Abfrage wird die Gültigkeit der Definitionsgleichung von h wirklich gefordert. Die Erhöhung von x wird also in der Schleifeninvarianten bereits berücksichtigt. Die Anpassung von h kann dann in der If-Anweisung konzentriert durchgeführt werden. Der Algorithmus sieht nach diesen Änderungen so aus:

```
x=0; y=r;
h= 3-2*r;
while (x<=y) {
  mark(x, y);
  x++;
  if (0<=h){y--; h+=4*(x-y)+2;} else h+= 4*x+2;
}
```

Demoprogramm

In der Lehrveranstaltung wird auf anspruchsvolle Programmierschnittstellen (Application Programming Interface, API) verzichtet. Die Grafikalgorithmen können mittels einfacher Blockgrafik demonstriert werden. Die Koordinaten des Zeichenbrett (x, y) mit den Farbwerten db[x][y] haben die in der Mathematik übliche Orientierung. Beim zeilenweisen Zeichnen mit der Funktion draw() wird der Punkt in der linken oberen Ecke des Zeichenbretts - das ist der Punkt mit den Koordinaten y=n-1, x=0 - zuerst gezeichnet. Die erste Zeile endet bei y=n-1, x=n-1.

Je Druckzeile werden zwei Punktzeilen ausgegeben. Jedes Druckzeichen wird also in eine obere und eine untere Hälfte aufgeteilt. Maßstabsgetreu ist das Druckergebnis demnach nur dann, wenn die Bildschirmzeichen genau doppelt so hoch wie breit sind. Andernfalls muss man den Blickwinkel ein wenig ändern, um zu dem gewünschten optischen Eindruck zu kommen.

Das vollständige Programm zur Demonstration des Algorithmus von Bresenham sieht so aus:

```
/*kreis.c, Timm Grams, Fulda, 04.01.02*/
#include <stdio.h>
#define m 79
#define n 50

int db[n][m];                                          /*Zeichenbrett*/
const int x0=m/2, y0=n/2;             /*(n, m) ist der Kreismittelpunkt*/

/*draw setzt voraus, dass die Zeichen doppelt so hoch wie breit sind
***********/
void draw() {
  int i, j;
```

```
  for(j=n-1; 0<j; j-=2) {
    printf("\n");
    for(i=0; i<m; i++) {
      if      (db[j][i]==0&&db[j-1][i]==0) printf("█");

      else if (db[j][i]==0&&db[j-1][i]==1) printf("▀");
      else if (db[j][i]==1&&db[j-1][i]==0) printf("▄");
      else if (db[j][i]==1&&db[j-1][i]==1) printf(" ");
    }
  }
}

void clip(int x, int y) {
  if (0<=x&&x<m&&0<=y&&y<n) db[y][x]=1;
}

void mark(int x, int y) {
  clip(x0+x, y0+y); clip(x0+y, y0+x);
  clip(x0+x, y0-y); clip(x0-y, y0+x);
  clip(x0+y, y0-x); clip(x0-x, y0+y);
  clip(x0-x, y0-y); clip(x0-y, y0-x);
}

void main(int argc, char *argv[]) {
  int x, y, h, r=10;
  if (argc>1) {
    sscanf(argv[1], "%d", &r);
  }

 /*Invariante:
   h = 2*x^2 + 4*x + 2*y^2 - 2*y + 3 - 2*r^2
 ****************************************/
 x=0; y=r;
  h= 3-2*r;
  while (x<=y) {
    mark(x, y);
    x++;
    if (0<=h){y--; h+=4*(x-y)+2;} else h+= 4*x+2;
  }


  draw();
}
```

Aufgaben

6.1 Beweisen Sie, dass der Initialisierungsteil des Programms die Invariante (*I*) tatsächlich gültig macht.

6.2 Beweisen Sie, dass der Kreisalgorithmus in der zuletzt angegebenen Version korrekt ist. Zeigen Sie zunächst, dass der Ausdruck

```
h = 2*x^2 + 4*x + 2*y^2 - 2*y + 3 - 2*r^2
```

tatsächlich eine Schleifeninvariante ist. Zeigen Sie dann, dass an der entscheidenden Stelle die Definitionsgleichung gilt.

6.3* Programmieren und testen Sie eine Funktion drawLine(), die eine Gerade in ein Raster zeichnet. Die ganzzahligen Koordinaten (x_0, y_0) und (x_1, y_1) der Endpunkte sind die Eingangsgrößen des Programms. Literaturhinweise: Stroustrup (1995, S. 195), Fellner (1988, S. 90 ff.), Wirth (1990).

7 Quicksort

Programmierstudie

Der Quicksort Algorithmus ist in nahezu allen Einführungstexten zur strukturierten Programmierung enthalten, z. B. in Alagic/Arbib (1978), Baber (1987), Gries (1981). Erfinder des Algorithmus ist C. A. R. Hoare, der auch die Korrektheit bewiesen hat (CACM-Algorithmus 323).

Dass der Zeitaufwand beim Quicksort-Algorithmus von der Größenordnung $n \log n$ ist, wird unter anderem im Buch von Aho/Hopcroft/Ullman (1983, S. 246 ff.) gezeigt. Er ist demnach viel effizienter als elementare Sortieralgorithmen wie Insertion Sort und Bubble Sort, deren Zeitaufwand in der Größenordnung von n^2 liegt.

Aufgabe: Wir betrachten ein Array a[m..n], für dessen Komponententyp eine vollständige Ordnung „<„ definiert ist. Die Komponenten dieses Arrays sind a[m], a[m+1], ..., a[n]. Der gesuchte Algorithmus soll das Array sortieren. Das heißt: Die Elemente des Arrays sind so anzuordnen, dass schließlich a[i]≤a[j] für alle i<j gilt.

Spezifikation: Es wird gleich eine nützliche Verallgemeinerung der Sortier-Funktion spezifiziert. Die Funktion

```
void sort(componentType a[], int l, int r) S
```

mit dem Funktionsrumpf *S* soll das Feld im Abschnitt a[l..r] sortieren und alle anderen Elemente an ihrem Platz lassen.

Für die genaue Spezifikation führen wir die Funktion *N* folgendermaßen ein: *N*(a, *e*) ist die Anzahl der Elemente *e* im Vektor a. Dass ein Feld b eine Permutation des Feldes a ist, lässt sich damit so ausdrücken: *N*(b, *e*) = *N*(a, *e*). Diese Gleichheit gilt für alle Elemente *e*, die in den Arrays a und b vorkommen - und für die übrigen sowieso[6].

Mit *A* bezeichnen wir den Wert des Feldes a unmittelbar vor Eintritt in die Prozedur; *A* ist Variable im mathematischen Sinn. Die formale Spezifikation des Prozedurrumpfs *S* lautet mit diesen Festlegungen so:

```
pre(S): a=A; m≤l≤r≤n

post(S): N(a, e)=N(A, e) für alle e; l≤μ<ν≤r impliziert a[μ]≤a[ν]
```

Das Semikolon in logischen Ausdrücken steht für das logische UND. Es bindet schwächer als alle anderen Verknüpfungen.

Methode: Es wir die Strategie des „Teile und herrsche„ verfolgt. Dabei wird das Problem so lange aufgeteilt, bis es verschwunden ist. Das geht so:

1. Auswahl eines Pivot-Elements (Pivot = Dreh-, Angelpunkt).
2. Aufteilen des Feldes in einen Teil, dessen Elemente nicht größer als das Pivot-Element sind und einen Teil, dessen Elemente nicht kleiner sind.
3. Beginne für jeden der Teile beim 1. Schritt (Rekursion).

Grobentwurf: Wir zerlegen die Gesamtaufgabe in die Teilaufgabe „Partition“ und „Sortieren der Teile“. Für das Sortieren der Teile steht die Prozedur sort zur Verfügung (Rekursion). Die

---

[6] Die Formulierung, dass eine Formel - sagen wir $f(e)$ - für alle möglichen Werte der Variablen *e* gilt, besagt, dass auf die Formel der Allquantor anzuwenden ist. Ausführlich würde man schreiben: $\bigwedge_e f(e)$.

Spezifikation von sort ist eingangs bereits erledigt worden. natürlich sind die formalen Parameter l und r in Vor- und Nachbedingung nun durch die aktuellen Parameter zu ersetzen.

```
void sort(componentTyp a[], int l, int r) {
  int i=l;
  partition;
  if (l<i-1) sort(a, l, i-1);
  if (i+1<r) sort(a, i+1, r);
}
```

Die Spezifikation des Programmabschnitts partition:

```
pre(partition): a=A; m≤l≤r≤n

post(partition): N(A, e)=N(a, e);  l≤i≤r; k<i impliziert a[k]≤a[i];
                                          i<k impliziert a[i]≤a[k]
```

In den Prädikaten der Nachbedingung von partition wird unausgesprochen vorausgesetzt, dass sie für alle möglichen Werte der Variablen *e* und *k* gelten. Anstelle von „i<*k* impliziert a[i]≤a[*k*]“ hätte man beispielsweise auch „(i<*k*) ≤ (a[i]≤a[*k*])“ schreiben können.

Der Korrektheitsnachweises für den Grobentwurf ist Gegenstand der Aufgabe 8.1

Was noch fehlt, ist die Konstruktion des Programmteils partition und der zugehörige Korrektheitsnachweis. Wir wollen hier auf die Konstruktionsarbeit verzichten und übernehmen die Lösung aus dem Buch von H. Schauer („PASCAL für Anfänger“, Oldenbourg, Wien 1979).

Mit einer lokalen Variablen g vom Elementtyp sieht der Programmteil partition so aus:

```
/*partition*/
i= l; j= r; g= a[i];
do {
  while (g<=a[j] && i<j) j--;
  a[i]= a[j];
  while (a[i]<=g && i<j) i++;
  a[j]= a[i];
} while (i<j);
a[i]= g;
```

Korrektheitsnachweis für den Programmteil partition:

Der Korrektheitsbeweis gestaltet sich etwas schwieriger als gewöhnlich, weil - anders als im Buch von Gries (1981) - die Restriktion auf die elementare Sortier-Operation Swap entfällt. Bei Gries stellt diese Restriktion sicher, dass stets nur Permutationen des ursprünglichen Felds entstehen. Die Grundidee des obigen Algorithmus ist, dass man im Feld ein „Loch“ aufmacht („g= a[i];“), das Platz für die Sortieroperationen schafft. Dieses Loch wandert über das Feld. Schließlich wird es - im Allgemeinen an einer anderen Stelle - wieder geschlossen („a[i]= g;“).

Um zu zeigen, dass nach Durchführung der Partition eine Permutation des ursprünglichen Arrays entsteht, zeigen wir, dass

```
N(A, e) = N(a, e) + ord(g=e) - ord(a[i]=e); m≤l≤i≤j≤r≤n
```

eine Invariante der Do-while-Schleife ist. Der erste Teil der Invariante besagt: Jedes Element *e* kommt im Array a und der Variablen g zusammengenommen genau so häufig vor wie im Array *A*. Ausnahme bildet das Element a[i]. Dieser Wert erscheint in a und g genau einmal zu oft[7].

---

[7] Man beachte, dass in mathematischen Ausdrücken das Gleichheitszeichen tatsächlich Gleichheit und nicht etwa Zuweisung bedeutet. Die Funktion ord angewandte auf boolesche Ausdrücke ist folgendermaßen definiert: ord(false)=0 und ord(true)=1.

Als erstes zeigen wir, dass die Invariante vor Schleifeneintritt wahr ist. Das läuft darauf hinaus, die Korrektheit der Spezifikation

pre(„i= l; j= r; g= a[i];"): a=*A*; m≤l≤r≤n

post(„i= l; j= r; g= a[i];"): *N*(*A*, *e*) = *N*(a, *e*) + ord(g=*e*) - ord(a[i]=*e*); m≤l≤i≤j≤r≤n

zu zeigen. Die Methode der symbolischen Auführung zeigt, dass nach Durchlaufen des Programms folgendes gilt: a=*A* und damit *N*(*A, e*) = *N*(a, *e*) für alle *e*. Außerdem ist g=*A*[i]. Wegen a[i]=*A*[i] ist der Wert ord(g=*e*) - ord(a[i]=*e*) gleichbedeutend mit ord(*A*[i]=*e*) - ord(*A*[i]=*e*), und der ist gleich null. Weil m, l, r und n im betrachteten Abschnitt nicht verändert werden und weil schließlich i=l und j=r ist, ist die Gültigkeit der Nachbedingung bewiesen. Vor dem Schleifeneintritt gilt also die Invariante.

Wir zeigen nun, dass die Invariante auch innerhalb der Do-while-Schleife an gewissen Stellen gilt. Der Programmabschnitt „while (g<=a[j] && i<j) j--; a[i]= a[j];" erhält die Invariante; genau genommen erfüllt er die folgende Spezifikation:

pre(„while (g<=a[j] && i<j) j--; a[i]= a[j];"):
N(*A*, *e*) = *N*(a, *e*) + ord(g=*e*) - ord(a[i]=*e*); m≤l≤i≤j≤r≤n

post(„while (g<=a[j] && i<j) j--; a[i]= a[j];"):
N(*A*, *e*) = *N*(a, *e*) + ord(g=*e*) - ord(a[i]=*e*); m≤l≤i≤j≤r≤n; a[i]=a[j]

Hinter der While-Schleife „while (g<=a[j] && i<j) j--;" ist die Invariante wahr, da außer j ja noch nichts verändert wurde und j auch nach der While-Schleife die gefordert Bedingung erfüllt.

Wir zeigen nun, dass auch die Zuweisung „a[i]= a[j];" die Gültigkeit der Invarianten aufrecht erhält. Wir führen eine mathematische Variable *B* ein und fügen den Teilausdruck a=*B* der Vorbedingung hinzu. Die Variable *B* soll die anfängliche Wertebelegungen festhalten. Da der Ausdruck a=*B* durch geeignete Wahl von *B* auf jeden Fall wahr gemacht werden kann, handelt es sich nicht um eine Verstärkung der Vorbedingung.

Da die Variablen m, l, i, j, r und m unverändert bleiben, schreiben wir den Teilausdruck „m≤l≤i≤j≤r≤n" nicht hin. Auch die Variable g bleibt unverändert. Für alle unveränderten Programmvariablen führen wir die jeweils zugehörigen mathematischen Variablen einfach dadurch ein, dass wir den Variablennamen kursiv schreiben.

pre(„a[i]= a[j];"): a=*B*; *N*(*A*, *e*) = *N*(a, *e*) + ord(g=*e*) - ord(a[*i*]=*e*)

post(„a[i]= a[j];"): *N*(*A*, *e*) = *N*(a, *e*) + ord(g=*e*) - ord(a[*i*]=*e*)

Aus der Vorbedingung erhält man folgende mathematische Beziehung

N(*A*, *e*) = *N*(*B*, *e*) + ord(*g*=*e*) - ord(*B*[*i*]=*e*)

Nach der Zuweisung „a[i]= a[j];" gelten die Gleichungen a[*i*]=a[*j*]=*B*[*j*]. Und für alle *k* ungleich *i* erhält man a[*k*]= *B*[*k*]. Das heißt: *N*(*B*, *e*) = *N*(a, *e*) + ord(*B*[*i*]=*e*) - ord(*B*[*j*]=*e*). Die Beziehungen bleiben auch dann gültig, wenn *i*=*j*.

Die mathematische Vorbedingung lässt sich mittels Äquivalenztransformationen und Rückübersetzungen so darstellen, dass schließlich die Nachbedingung entsteht:

N(*A*, *e*) = *N*(*B*, *e*) + ord(*g*=*e*) - ord(*B*[*i*]=*e*)

N(*A*, *e*) = *N*(a, *e*) + ord(*B*[*i*]=*e*) - ord(*B*[*j*]=*e*) + ord(*g*=*e*) - ord(*B*[*i*]=*e*)

N(*A*, *e*) = *N*(a, *e*) - ord(*B*[*j*]=*e*) + ord(*g*=*e*)

N(*A*, *e*) = *N*(a, *e*) + ord(*g*=*e*) - ord(a[*i*]=*e*)

N(*A*, *e*) = *N*(a, *e*) + ord(g=*e*) - ord(a[i]=*e*); m≤l≤i≤j≤r≤n; a[i]=a[j]

Das war zu zeigen. Der letzte Ausdruck impliziert den Ausdruck

```
N(A, e) = N(a, e) + ord(g=e) - ord(a[j]=e); m≤l≤i≤j≤r≤n
```

Diesen nehmen wir als Vorbedingung für den Folgeabschnitt des Programms.

```
pre(„while (a[i]<=g && i<j) i++;  a[j]= a[i];"):
  N(A, e) = N(a, e) + ord(g=e) - ord(a[j]=e); m≤l≤i≤j≤r≤n
post(„while (a[i]<=g && i<j) i++;  a[j]= a[i];"):
  N(A, e) = N(a, e) + ord(g=e) - ord(a[i]=e); m≤l≤i≤j≤r≤n; a[i]=a[j]
```

Der Korrektheitsnachweis bietet gegenüber dem vorhergehenden nichts Neues. Insgesamt gilt also, dass der Programmabschnitt partition tatsächlich nur eine Permutation des Arrays a erzeugt. Damit ist der Korrektheitsnachweis für den Programmteil partition noch nicht vollständig. Die Vervollständigung wird als Aufgabe 8.2 gestellt.

Hier ist der komplette *Quicksort-Algorithmus:*

```
void sort(componentTyp a[], int l, int r) {
  /*partition*/
  int i= l, j= r;
  componentTyp g= a[i];
  do {
    while (g<=a[j] && i<j) j--;
    a[i]= a[j];
    while (a[i]<=g && i<j) i++;
    a[j]= a[i];
  } while (i<j);
  a[i]= g;
  /*ende partition*/
  if (l<i-1) sort(a, l, i-1);
  if (i+1<r) sort(a, i+1, r);
}
```

Exkurs: Erfinden und Optimieren von Algorithmen

Hier geht es um Lösungsfindeverfahren, also um Techniken, die einen auf neue Ideen bringen können. Solche Lösungsfindeverfahren - ein Art von Faustregeln - werden in der Kognitionspsychologie Heuristiken genannt. Mit den Worten Heinrich von Pierers fragen wir: „Wie kommt das Neue in die Welt?" Genauer: Wie können wir Denkblockaden aufbrechen, wie den Einstellungseffekt überwinden.

Bewusste Aktivierung von Heuristiken

Eine Möglichkeit ist die *bewusste Aktivierung von Heuristiken*. In „Denkfallen und Programmierfehler" (Grams, 1990) sind - in Anlehnung an die berühmte SCHULE DES DENKENS (Engl.: HOW TO SOLVE IT) von Georg Polya - die wichtigsten Dinge zusammengestellt. Wir starten mit der

Basisheuristik: Kann ich in der Liste von Heuristiken eine finden, die mir weiterhilft?

Und hier ist die Liste der Heuristiken

1. Heuristik: *Analogie*

Habe ich etwas Ähnliches schon gesehen? Kenne ich ein verwandtes Problem?

2. Heuristik: *Verallgemeinerung*

Bringt mich der Übergang von einem Objekt zu einer ganzen Klasse von Objekten weiter?

Eine Möglichkeit bietet das so genannte *Einbettungsprinzip*. Es wird für Algorithmen auf Netzen und in der Dynamischen Optimierung genutzt. In „Denkfallen und Programmierfehler“ habe ich das Prinzip am Beispiel eines Worterkennungsalgorithmus erläutert.

3. Heuristik: *Spezialisierung*

Komme ich weiter, wenn ich erst einmal einen leicht zugänglichen Spezialfall löse?

4. Heuristik: *Variation*

Kann ich durch Veränderung der Problemstellung der Lösung näher kommen? Kann das Problem anders ausdrücken?

5. Heuristik: *Rückwärtssuche*

Ich sehe mir das gewünschte Ergebnis an. Welche Operatoren könnten mich zu dem Ergebnis führen?

6. Heuristik: *Teile und herrsche (divide et impera)*

Lässt sich das Problem in leichter lösbare Teilprobleme zerlegen?

Beispiel: Quicksort

7. Heuristik: *Vollständige Enumeration*

Kann ich mir (alle) Lösungen verschaffen, die wenigstens einen Teil der Zielbedingungen erfüllen?

Auf meiner Web-Seite finden Sie eine Lektion über „Schöpferisches Denken - Heuristik“. Sie bietet eine Reihe von Denkaufgaben, mit denen Sie die Technik der bewussten Aktivierung von Heuristiken einüben können.

Regeln zur Feinabstimmung von Algorithmen

Neben diesen Heuristiken legt man sich besser auch noch ein paar eher konkrete Regeln zurecht, die einem bei der *Feinabstimmung* (Fine-Tuning) von Algorithmen leiten. Diese Regeln dienen in erster Linie der Verbesserung der Zeiteffizienz. Damit verbunden ist meist auch eine textliche Reduzierung. Die Regeln ergänzen die Programmierregeln des 3. Abschnitts („Programmieren nach Regeln“).

Die durch Befolgung der Regeln erreichte Komprimierung macht das Programm meist eleganter, aber auch unverständlicher. Daraus ergibt sich der Bedarf an zusätzlichen Kommentaren. Folglich sind auch die Regeln der Kommentierung zu ergänzen. Die Grundprinzipien der Sparsamkeit und der Prägnanz der Kommentierung stehen weiterhin ganz obenan.

1. Feinabstimmungsregel: *Geschachtelte Schleifen vermeiden.*

Beispiel ist das Wortsuche-Programm: Intuitive Entwürfe haben meist zwei geschachtelte Schliefen. Im 5. Abschnitt wird gezeigt, wie man die Sache mit einer einzigen Schleife erledigen kann. Das Programm ist effizienter als Zwei-Schleifen-Versionen.

2. Feinabstimmungsregel: *Berechnungen vereinfachen.*

Wiederholte und komplizierte Berechnungen können oft durch das Einführen einer weiteren Variablen anstelle des komplexen Ausdrucks reduziert werden. Das Invariantenkonzept hilft bei Entwurf und Korrektheitsbeweis (Definitionsgleichung = Invariante). Das wurde in den Abschnitten 5 und 6 demonstriert.

3. Feinabstimmungsregel: *Mehrfacher Änderungen einer Variablen innerhalb einer Schleifenanweisung vermeiden.*

Beispiel dafür ist die Feinabstimmung des Kreisalgorithmus am Schluss des 6. Abschnitts.

4. Feinabstimmungsregel: *Schleifen kommentieren.*

Unmittelbar vor einer Schleife bzw. deren Initialisierungsteil sollte man die wichtigsten und aussagekräftigsten Schleifeninvarianten angeben. Grundsätzlich sind Kommentare kompakt zu schreiben und so anzuordnen, dass der Programmtext nicht zerrissen wird, analog zur Kommentierung von Funktionen. Vor dem Kommentar und hinter der Schleife sollte man je eine Leerzeilen einfügen. So wird sichtbar, was zusammengehört. Das ist im Demoprogramm des Kreisalgorithmus so gemacht worden (6. Abschnitt).

Aufgaben

7.1 Führen Sie einen (nicht formalen) anschaulichen *Korrektheitsnachweis* für den Grobentwurf durch. Zeigen Sie, dass der Korrektheitsnachweis durch vollständige Induktion über r-l formalisiert werden kann. Siehe auch Alagic/Arbib (1978, S. 200 f.).

7.2 Vervollständigen Sie den Korrektheitsbeweis für den Programmteil partition. *Hinweis:* Das Prädikat „k<i impliziert a[k]≤g; j<k impliziert g≤a[k]“ ist Invariante sowohl der Repeat-Schleife als auch der beiden While-Schleifen.

8 Fortgeschrittene Programmiermethoden

Die Methode von Hoare

Zwei wesentliche Mängel hat die bisher verwendete Methode der Programmbeweise:

1. Eine recht umständliche Notation (Nebeneinander von mathematischen Variablen und Programmvariablen) und fehleranfällige Verfahrensweise (separate Ermittlung und Berücksichtigung von Programmpfaden), sowie
2. eine schwache Unterstützung der Programmierung auf der Basis von Korrektheitsbeweisen.

Erläuterung zu Punkt 2: Die Beweisführung bei der symbolischen Methode geschieht in Richtung des Programmablaufs - also von vorn nach hinten. Demgegenüber muss die Konstruktion mit der Spezifikation beginnen. Da die wesentlichen Aussagen der Spezifikation in der Nachbedingung stecken, müsste die Konstruktion eigentlich von hinten beginnen. Gefordert ist also das „Denken vom Resultat her".

Die Methode von Hoare überwindet die Schwierigkeiten und sie unterstützt das Denken vom Resultat her. Außerdem ist sie *formal*. Demgegenüber war unsere bisherige Beweisführung nur eine Art *strenger Argumentation*[8].

Beweisregeln

Bei der Methode von Hoare geht grundsätzlich darum, zu einer gegebenen Nachbedingung *R* einer Anweisung *S* eine möglichst schwache Vorbedingung zu bestimmen. Diese wird mit wp(*S*, *R*) bezeichnet (wp steht für weakest precondition). Die schwächste Vorbedingung ist eine Bedingung, die von allen anderen Vorbedingungen, für die die Anweisung korrekt ist, impliziert wird. Genauer: Eine Anweisung *S* ist genau dann korrekt bezüglich der Spezifikation pre(*S*) und post(*S*), wenn die Bedingung

$$pre(S) \leq wp(S, post(S)) \qquad \text{(Korrektheitsbedingung)}$$

erfüllt ist. Die Bedingung ist also sowohl notwendig als auch hinreichend für die Korrektheit[9].

Für die Beweisführung gibt es eine Reihe von Axiomen, das sind Regeln, nach denen die jeweilige schwächste Vorbedingung aus der Nachbedingung und der Anweisung ermittelt werden kann - zum Beispiel die folgende *Zuweisungsregel:*

$$wp(„x = e;", R) = R^x_e \qquad \text{(Zuweisungsregel)}$$

---

[8] Wir unterscheinden hinsichtlich der Strenge der Beweisführung zwischen *formalem Beweis* und *strenger Argumentation*. Die Methode von Hoare gehört zu den formalen Beweismethoden. Die strenge Argumentation entspricht der Beweisführung in der Mathematik (Donald MacKenzie: "Computers, Formal Proofs, and the Law Courts". Notices of the American Mathematical Society. 39 (1992) 9, pp. 1066-1069). Der formale Beweis führt zu schwerer lesbaren Texten als die strenge Argumentation. Andererseits ist die strenge Argumentation fehleranfälliger. Aber indem ein nicht formaler Beweis "immer wieder der Überprüfung und Beurteilung eines neuen Publikums ausgesetzt wird, wird er einem dauernden Prozeß der Kritik und Neubewertung unterworfen. Irrtümer, Unklarheiten und Mißverständnisse werden durch diesen dauernden Durchleuchtungsprozeß aufgeklärt" (Davis, P. J; Hersh, R.: Erfahrung Mathematik. Birkhäuser, Basel, 1985, 150 ff.).

[9] Das ≤-Zeichen steht für die Implikation: „A≤B" heißt soviel wie, „Wenn A dann B". Da üblicherweise der Wahrheitswert „falsch" mit 0 und „wahr" mit 1 identifiziert wird, ist diese Festlegung in Übereinstimmung mit der Kleiner-oder-gleich-Beziehung für Zahlen.

Also: Man erhält die Vorbedingung einer Zuweisung x= e mit der Nachbedingung *R*, indem man überall in der Nachbedingung die Variable x durch den Ausdruck *e* ersetzt. Dabei ist *e* ein mit x zuweisungsverträglicher Ausdruck ohne Nebenwirkungen.

Herleitung der Zuweisungsregel: Sei σ ein Zustand der Variablen des Programms (Zustand = Wertebelegung) und *a* ein möglicher Wert der Variablen x. Mit σ[x/*a*] bezeichnen wir dann denjenigen Zustand, der sich vom Zustand σ höchstens im Wert der Variablen x unterscheidet und es gilt

σ[x/*a*](x) = *a* und

σ[x/*a*](y) = σ(y) für alle Variablen y ungleich x.

Die Bedeutung der nebenwirkungsfreien Zuweisung ist - unter der Bedingung, dass der Ausdruck *e* im Zustand σ überhaupt einen Wert liefert - bekanntlich folgendermaßen definiert:

„x = *e*;"(σ) = σ[x/*e*(σ)]

Wenn *R* die Nachbedingung der Zuweisung ist, dann muss $R(\sigma[x/e(\sigma)])$ wahr sein.

Offenbar gilt $R(\sigma[x/e(\sigma)]) = R^x_e(\sigma)$. Also muss R^x_e tatsächlich von jedem Zustand σ erfüllt sein, wenn die Zuweisung das erwünschte Resultat *R* haben soll. Jede andere zulässige und korrekte Vorbedingung muss daher R^x_e implizieren. Also ist R^x_e tatsächlich die gesuchte schwächste Vorbedingung der Zuweisungsanweisung.

Die Zuweisungsregel gilt zunächst nur im Sinne der partiellen Korrektheit. Für die vollständige Korrektheit ist die Vorbedingung gegebenenfalls um eine weitere Bedingung zu ergänzen, die sicherstellt, dass die Auswertung des Ausdrucks *e* stets zu einem definierten Ergebnis führt.

Achtung bei Arrays: Bei naiver Anwendung der Zuweisungsregel auf ein Array b[0..n] könnte man darauf kommen, dass wp(„b[i]=5;", b[i]=b[j]) gleich $[b[i]=b[j]]^{b[i]}_5$ ist, und dass daraus die Formel b[j]=5 für die schwächste Vorbedingung folgt. Aber im Falle i=j muss diese Vorbedingung keineswegs erfüllt sein. Dennoch gilt schließlich die Nachbedingung b[i]=b[j]. Für Arrays muss man die Zuweisungsregel immer auf das gesamte Array beziehen. Sei (b; i:*e*) die Bezeichnung für ein Array, das aus b dadurch entsteht, dass Komponente b[i] Ergebnis des Ausdrucks *e* ist. Alle anderen Komponenten sind unverändert:

(b; i:*e*)[j] = *e* für j=i und (b; i:*e*)[j] = b[j] sonst.

Jetzt findet man die korrekte schwächste Vorbedingung:

wp(„b[i]=5;", b[i]=b[j])

= $(b[i]=b[j])^{b}_{(b;\ i:5)}$

= ((b; i:5)[i] = (b; i:5)[j])

= (5= (b; i:5)[j])

Jetzt sieht man: Die Vorbedingung 5= (b; i:5)[j] ist erfüllt, wenn j=i ist oder wenn b[j]=5 ist. Auch hier gilt: Der schwächsten Vorbedingung ist um Bedingungen zu verstärken, so dass die Auswertung aller beteiligten Ausdrücke möglich ist und dass die Indizes aus dem Indexbereich des Arrays sind.

Weitere Beweisregeln aus dem System von Hoare werden hier nur kurz angegeben. Als erstes eine Regel, die es erlaubt, Programmbeweise in einfache Teilbeweise aufzugliedern:

wp(*S*, R_1) $\wedge$ wp(*S*, R_2) = wp(*S*, $R_1 \wedge R_2$) (Konjunktionsregel)

Im Zuge des Programmbeweises für den Quicksort-Algorithmus haben wir diese Konjunktionsregel bereits benutzt. Für eine Sequenz der Anweisungen *S* und *T*, kurz *ST*, gilt

$wp(ST, R) = wp(S, wp(T, R))$ (Sequenzregel)

Und die Semantik der Auswahlanweisung führt auf

$wp(\text{„if}(B)\ S_1\ \text{else}\ S_2\text{“}, R) = B \wedge wp(S_1, R) \vee \neg B \wedge wp(S_2, R)$ (Auswahlregel 1)

Eine äquivalente Formulierung der Auswahlregel ist

$wp(\text{„if}(B)\ S_1\ \text{else}\ S_2\text{“}, R) = (B \leq wp(S_1, R)) \wedge (\neg B \leq wp(S_2, R))$ (Auswahlregel 2)

Auch der bereits behandelte Schleifensatz lässt sich nun mit Hilfe der schwächsten Vorbedingungen neu formulieren.

Wenn ein Prädikat I die Endlichkeit der While-Schleife „while (B) S“ impliziert und außerdem die Invarianteneigenschaft $(I \wedge B) \leq wp(S, I)$ besitzt, dann gilt
$I \leq wp(\text{„while}\ (B)\ S\text{“}, I \wedge \neg B)$. (Schleifensatz)

Ein einfaches Beispiel

An einem sehr einfachen Beispiel soll die Vorgehensweise des beweisgeleiteten Programmierens verdeutlicht werden[10]. Wir nehmen an, dass das zu entwerfende Programmstück *S* die Variablen a, b, c (vom ganzzahligen oder reellen Typ) enthält. Es soll folgende Spezifikation erfüllen:

$pre(S)$: true

$post(S)$: $(c=a+b) \wedge (a=2) \wedge (b=5)$

Wir zerlegen die Anweisung S in zwei Teile, von denen der zweite eine Zuweisung ist:

S: S_1; c= a+b;

und bilden die schwächste Vorbedingung der Zuweisungsanweisung:

$wp(\text{„c= a+b;“}, (c=a+b) \wedge (a=2) \wedge (b=5))$

$= [(c=a+b) \wedge (a=2) \wedge (b=5)]^{c}_{a+b}$

$= (a+b=a+b) \wedge (a=2) \wedge (b=5)$

$= \text{true} \wedge (a=2) \wedge (b=5)$

$= (a=2) \wedge (b=5)$

Daraus folgt, dass S_1 die Spezifikation

$pre(S_1)$: true

$post(S_1)$: $(a=2) \wedge (b=5)$

erfüllen muss.

Wir haben nun die Konstruktion des Programmabschnitts S_1 zu erledigen. Naheliegend ist folgender Realisierungsvorschlag: S_1 = „a= 2; b= 5;“. Die Korrektheit lässt sich wieder mit der Zuweisungsregel - Schritt für Schritt von hinten nach vorne gehend - nachweisen.

$wp(\text{„b= 5;“}, (a=2) \wedge (b=5))$

---

[10] Eine vorzügliche Einführung in das *beweisgeleitete Programmieren* bietet das Lehrbuch „The Science of Programming“ von David Gries (1981). Umfassend dargestellt ist Hoares Beweistechnik auch im Standardwerk „The Design of Well-Structured and Correct Programs“ von Alagic und Arbib (1978). Eine eher rezeptbuchartige Zusammenfassung bietet der „Software-Zauberlehrling“ von Robert Baber (1990). In meinem Buch „Denkfallen und Programmierfehler“ (1990) stelle ich den Zusammenhang zwischen allgemeinen Problemlösungsprozessen und dem Programmieren her und erkläre auch ausführliche, warum diese Art des Programmierens „semi-algorithmisch" genannt werden kann.

```
= [(a=2)∧(b=5)]^b_5
= (a=2)∧(5=5)
= (a=2)
```

Und weiter:

```
wp(„a= 2;", a=2)
= [a=2]^a_2
= (2=2)
= true
```

Jetzt ist das Programm *S* konstruiert. Es sieht so aus:

```
a= 2; b= 5; c= a+b;
```

Und gleichzeitig ist die Korrektheit des Programms bewiesen.

Die Methode realisiert streng das „Denken vom Resultat her": Programmentwurf und Beweis gehen schrittweise von hinten nach vorn und - bei geschachtelten Strukturen wie Prozeduren, Schleifen, Verzweigungen - von außen nach innen. Das ist der Kern der *strukturierten Programmierung*.

Anwendungen und Erfahrungen

Ich habe mit der Methode von Hoare gute Erfahrungen gemacht - nicht nur beim *Programmieren im Kleinen*, sondern auch beim *Programmieren im Großen*. Formale Methoden wie die von Hoare finden man heute vor allem in der Sicherheitstechnik und in der Prozessautomatisierung.

Beispiel 1: Die Goldversionen der Programme zu den Programmierstudien habe ich nach der Methode von Hoare entwickelt. Dabei habe ich bewusst auf den Editor und den Debugger der Programmierumgebung verzichtet und die Programme mit einem Textverarbeitungsprogramm entwickelt. Die Programme waren von Anfang an fehlerfrei.

Beispiel 2: Der obige Nachweis der Korrektheit des Quicksort-Algorithmus habe ich zunächst mit der Methode von Hoare geführt. Erst für die Lehrveranstaltung habe ich den Beweis auf die symbolische Ausführung umgestellt.

Beispiel 3: Der CACM-Algorithmus 246 zur Erstellung von Gray-Codes ist aus dem Programmtext heraus nicht zu verstehen. Ich habe ein eigenes Programm für den Gray-Code geschrieben und zunächst an einigen Beispielen gezeigt, dass das CACM-Programm und mein eigenes vermutlich äquivalent sind. Meine Version war langsamer. Dann habe ich mein Programm Schritt für Schritt in die CACM-Version umgewandelt. Dass es sich bei jedem der Umformungsschritte um eine Äquivalenztransformation des Programmes gehandelt hat, habe ich durch eingestreute Zusicherungen und Korrektheitsnachweise sichergestellt. Es stellte sich heraus, dass das CACM-Programm noch eine überflüssige Anweisung enthielt.

Beispiel 4: Im „Verkehrsmodell der Stadt Fulda" - es ist in Object Pascal programmiert - wurden die verschiedenen Programmschichten unabhängig voneinander programmiert. Als in das ansonsten fast fertige Programm das Modul für die Ampelsteuerung eingefügt wurde, kam es zu Fehlern: Das Programm stieg aus, weil die Aktivierung von Fahrstreifen unterblieb. Reparaturversuche nach der Trial-and-Error-Methode führten nur zu Verlagerungen des Problems. Schließlich wurde der Fehler flächendeckend mittels einer Invariante für die Lebendigkeit von Fahrstreifen beseitigt: Nachdem die problemlösende Invariante formuliert war, konnten alle relevanten Prozeduren im Hinblick auf die Aufrechterhaltung der Invarianten überarbeitet werden. Auch in diesem Fall hat die Methode Zeit und Nerven gespart.

Über den Gebrauch von integrierten Programmierumgebungen

Heute gibt es weit entwickelte integrierte Programmierumgebungen (Integrated Development Environment, IDE). Sie ermöglichen einen einheitlichen Zugang zu den Entwicklungshilfsmitteln (Tools) einer Sprache:

- Text-Editor, möglichst mit Syntax-Prüfung bereits während der Eingabe,
- Editor für die Gestaltung von Bedienoberflächen (Graphical User Interface, GUI) durch direkte Manipulation der Grafikelemente (Drag-and-Drop-Technik),
- Compiler,
- Debugger,
- Projekt- und Versionenverwaltung,
- Laufzeitumgebung.

Solche Werkzeuge sind für eine leistungsfähige Software-Produktion heute unentbehrlich. Die Hilfsmittel verleiten den Anfänger aber auch dazu, das Programmieren als Bastelaufgabe zu sehen: Das Programm wird, nachdem man die Aufgabe gerade mal verstanden zu haben glaubt, in die Maschine gehackt. Die direkte Manipulation der Buttons und Panels ermutigt dazu, auf ein akribisches Layout der Bedienoberfläche zu verzichten. Unter Zuhilfenahme des meist sehr leistungsfähigen Debuggers wird dann die meiste Zeit der Programmentwicklung mit Fehlerbeseitigung verbracht. Und schließlich wird die Spezifikation dem erreichten Entwicklungsstand angepasst. So entsteht Software minderer Qualität:

- Der Lösungsprozess wird zu Hauptsache und die gestellte Aufgabe verschwindet aus dem Blickfeld. Das Ziel wird möglicherweise nicht erreicht.
- Mancher Fehler verbleibt im Programm.

Bei einer sorgfältigen Programmentwicklung von Anfang an lassen sich die Fehler vermeiden. Ich empfehle, immer dann, wenn eine kitzlige Programmieraufgabe zu lösen ist, *auf die integrierte Entwicklungsumgebung ganz zu verzichten*.

Das fällt nicht sehr schwer: Beim Algorithmenentwurf werden nur relativ wenige Elemente der Programmiersprache benötigt. Eine fortlaufende Kontrolle auf korrekte Syntax ist daher entbehrlich. Und einen Debugger braucht man beim beweisgeleiteten Programmieren schon gar nicht.

Am besten entwickelt man Programm und Beweis Hand in Hand. Das kann man mit jedem Textsystem machen. Wenn man das Programm beisammen hat, kopiert man es in eine reine Textdatei, ergänzt die Ein-/Ausgabe-Anweisungen und fügt die nötigen Includes hinzu. Dann lässt man das Programm laufen. Meine Erfahrung ist, dass Programme, die so entwickelt worden sind, tatsächlich auf Anhieb laufen und ausschließlich korrekte Ergebnisse liefern.

Aufgaben

8.1 *Plateau:* Sei `a` ein sortiertes Integer-Array „`int a[n];`“. Es gilt also `a[i]≤a[j]` für alle `i<j<n`. Gesucht ist die Länge `p` des längsten Plateaus (Folge gleicher Werte). Der Programmentwurf ist nach der Methode von Hoare durchzuführen.

8.2*** *Generatorpolynom:* Es ist der Exponent eines Polynoms über dem Körper der Binärzeichen zu ermitteln. Der Exponent e ist die kleinste Zahl derart, dass die e+1-stellige Folge 100...01 durch das Generatorpolynom teilbar ist. Beispiel: Das Polynom $1+X^2+X^3$ lautet in Folgenschreibweise 1011. Das Divisionsschema liefert

```
10000001 = 1011*111
--------
1011
  1011
   1011
    1011
```

Der Exponent des Polynoms 1011 ist also gleich 7.

Es ist ein Programm nach der Methode von Hoare zu entwickeln, das für jedes beliebig vorgebbare Generatorpolynom den Exponenten liefert.

9 Computerarithmetik

Fehlerarten und deren Kontrolle

Wenn ein Programm nicht das tut, was es soll, dann kann das folgende Ursachen haben

1. Konzeptfehler (aufgrund falscher Vorstellungen vom zu lösenden Problem beispielsweise)
2. Entwurfsfehler (Fehler beim Übergang von der Idee zur Architektur)
3. Programmierfehler (Implementierungs- und Realisierungsfehler)
4. *Rundungsfehler* (aufgrund der Computerarithmetik)
5. *Verfahrensfehler* (des mathematischen Verfahrens) und *Fehlerfortpflanzung*
6. Datenfehler (fehlerhafte Eingabedaten)

In den bisherigen Kapiteln stand die Vermeidung der Fehlertypen 2 und 3 im Zentrum des Interesses. Rundungsfehler (4) spielten bislang keine Rolle, da wir uns bei den Zahlen auf die ganzen Zahlen beschränkt haben. Und auch die Verfahrensfehler (5) machen sich erst bemerkbar, wenn es um die Lösung von Problemen der Analysis geht, beispielsweise die numerischen Behandlung von Differentialgleichungen.

Für die *Computerarithmetik* der reellen Zahlen gibt es leider keine Entsprechung zur Logik mit ihren mächtigen Äquivalenztransformationsregeln. Hier sind von Fall zu Fall auf mathematischem Wege Fehlerabschätzungen durchzuführen und dann in das Programm zu übertragen.

Beispiel: Gesucht ist der Fixpunkt der Funktion $f(x) = e^{-x}$, also die Zahl x für die $f(x) = x$ gilt.

Die Funktion g sei gegeben durch $g(x)=x-f(x)$. Diese Funktion ist stetig und es gilt $g(1/2) = 1/2 - e^{-1/2} < 0$ und $g(1)=1-1/e > 0$. Nach dem Zwischenwertsatz hat die Funktion im Intervall [½, 1] eine Nullstelle. Und das heißt, dass die Funktion f in diesem Intervall einen Fixpunkt ξ besitzt.

Die Funktion f ist auf dem Intervall *kontrahierend*. Das heißt: Es gibt eine nichtnegative Konstante[11] L derart, dass $L<1$ und dass $|f(x)-f(y)|\leq L\cdot|x-y|$ für alle $x, y \in [½, 1]$. Wir wählen als Konstante L einfach den Maximalwert, den der Betrag der ersten Ableitung der Funktion f' auf dem Intervall annimmt. Hier gilt konkret $L=|f'(½)|= e^{-½} \approx 0.6065306$.

Da die Ableitung der Funktion auf dem Intervall [½, 1] betragsmäßig kleiner als eins ist, sind die Voraussetzungen des *Fixpunktsatzes* erfüllt (Stoer, 1994, S. 292). Die Iteration $x_{n+1} = f(x_n)$ führt mit einem geeigneten Anfangswert (beispielsweise $x_0=½$) wegen

$$| x_n - \xi | = | f(x_{n-1}) - f(\xi) | \leq L | x_{n-1} - \xi | \leq L^n | x_0 - \xi |$$

auf eine gegen den Fixpunkt ξ konvergierende Folge $x_0, x_1, x_2, ...$

Die Iteration lässt sich einfach in ein Programm übertragen:

```
float f(float x) {return exp(-x);}

void main(){
  float x=1, y=0;
  while (x!=y) x=f(y=x);
  printf("\n! x = %g", x);
}
```

[11] auch: Lipschitz-Konstante

Aber so funktioniert das nicht: Die Iteration kommt nicht zum Ende. Im Bereich der Gleitpunktzahlen hat *f keinen* Fixpunkt. Es wurde vergessen, den Rundungsfehler in Betracht zu ziehen. Das wird verbessert:

```
const float eps= FLT_EPSILON;
...
while (fabs(x-y)>=fabs(x)*eps) { ... }
```

Dabei ist der Wert eps eine Abschätzung des relativen Rundungsfehlers der Gleitpunktdarstellung `float` nach ANSI/IEEE-Standard.

Richtig gut ist diese Lösung aber noch nicht: Wenn die Funktion aus einer ganzen Folge von Rechenschritten besteht, können wir gar nicht sicher sein, dass der relative Fehler der Berechnung durch das gewählte ε abgedeckt wird. Außerdem wird in die Funktion `f` ja bereits ein fehlerbehafteter Wert eingesetzt. Eine *verifizierte Berechnung* verlangt, dass alle diese Fehler kontrolliert werden.

Ein praktikabler Weg zum nachweisbar korrekten Programm mit Gleitpunkt-Arithmetik ist die Verwendung der Intervallarithmetik. Sie erlaubt die Kontrolle von Rundungs- und Verfahrensfehlern. In den folgenden Abschnitten werden die Grundideen vorgestellt.

Auch wenn der Programmierer später nicht mit der Intervallarithmetik in Berührung kommt, ist eine Beschäftigung mit dem Thema sehr lehrreich. Die Intervallarithmetik stellt einen Versuch dar, die „Fehlerquelle Computer“ zu beherrschen. Sie verdeutlicht, was alles schief gehen kann.

Die Zahlendarstellungen `float` *und* `double`

Zahlendarstellungen nach ANSI/IEEE Standard 754-1985: Der Wert der Gleitpunktzahl im Stellenwertsystem zur Basis 2 ist gegeben durch

$$z = m \cdot 2^e = (-1)^v \, |m| \cdot 2^e$$

Dabei ist e eine ganze Zahl, und m ist eine (vorzeichenbehaftete) Festpunktzahl mit t Stellen nach dem Dezimalpunkt. Beiden Zahlendarstellungen wird das Stellenwertsystem zur Basis 2 zugrundegelegt. Der Exponent e liegt in Exzessdarstellung mit dem Exzess q vor. Bei positivem Vorzeichen ist v=0 und bei negativem ist v=1.

Die Zahlendarstellung wird *normalisiert* genannt, wenn

$$1 \le |m| < 2 \qquad (*)$$

ist. In diesem Fall hat die die Zahl $|m|$ die Form $|m| = (c_0.c_{-1}c_{-2}...c_{-t})_2$ mit c_0=1. Rechnerintern wird die Zahl nach dem Prinzip „Das Wichtigste zuerst“ gespeichert:

v	$e+q$	$(\|m\|-1)\cdot 2^t$

Die *gespeicherten Mantissenwerte* sind $(|m|-1)\cdot 2^t = (c_{-1}c_{-2}...c_{-t})_2$. Die Parameter q und t der Datentypen `float` und `double` sind in der Tabelle 9.1 gegeben.

Der Exzess q ist eine Zweierpotenz minus 1: $q=2^k-1$. Die Exzessdarstellung ist $(k+1)$-stellig. Für die Codezeichen $e+q$ stehen also die Zahlen von 0 bis $2^{k+1}-1$ zur Verfügung. Die Exponenten e (ursprüngliche Zeichen) reichen also von $-(2^k-1)$ bis 2^k. Der kleinste und der größte Wert stehen für die Darstellung sehr kleiner und sehr großer - nicht normalisierter - Zahlen und für Fehlermeldungen zur Verfügung. Der kleinste Exponent der normalisierten Gleitpunktdarstellung ist also $e_{min}=-(2^k-2)$ und der größte ist $e_{max} = 2^k - 1$.

Genauer werden folgende Bereiche unterschieden:

$0 < e+q < 2^{k+1}-1$: normalisierte Gleitpunktdarstellung mit $e_{min} \le e \le e_{max}$.

$0 = e+q$: Nicht-normalisierte Darstellung sehr kleiner Zahlen. Es ist $z = m \cdot e^{e_{min}}$ und bei $|m| = (c_0.c_{-1}c_{-2}...c_{-t})_2$ wird $c_0=0$ vorausgesetzt. Wenn alle Bits der rechnerinternen Darstellung der Mantisse gleich 0 sind, handelt es sich um die Zahl Null: ±0. Bei der nicht-normalisierten Zahlendarstellung reduziert sich die Stellenzahl je kleiner die Zahl wird. Die relative Genauigkeit nimmt also ebenfalls ab.

$e+q = 2^{k+1}-1$: Die rechnerinterne Darstellung des Exponenten besteht aus lauter Einsen. Wenn die gespeicherten Mantissenwerte alle gleich null sind, handelt es sich um die Darstellung des vorzeichenbehafteten Unendlich: ±∞. Hat die gespeicherte Mantisse Werte ungleich null, handelt es sich nicht um eine Zahlendarstellung (Not a Number, NaN), sondern um die Darstellung einer Fehlermeldung im Anschluss an eine ungültige Gleitpunktoperation wie eine Division durch null.

Rundungsfehler

Die Genauigkeit der Gleitpunktdarstellung wird durch die folgenden Zahlen erfasst:

- min kleinste positive Maschinenzahl
- eps Obergrenze des relativen Rundungsfehlers (maximaler relativer Abstand benachbarter Gleitpunktzahlen)

Mit t Nachpunktstellen und einem Exzesswert von q und unter Zugrundelegung der Zahlendarstellung nach dem ANSI/IEEE-Standard erhält man die Formeln

$\text{min} = 2^{-(q-1)} \cdot 2^{-t}$ und $\text{eps} = 2^{-t}$.

Für die Gleitpunktdarstellungen `float` und `double` sind die Werte in der Tabelle 9.1 zusammengestellt.

	`float`	**`double`**
t	23	52
q	127	1023
eps	1.192093_{10}-7	2.220446_{10}-16
min	1.401298_{10}-45	4.940656_{10}-324

Tabelle 9.1 Genauigkeit der Gleitpunktdarstellung

Der Zusammenhang zwischen der Gleitpunktdarstellung rd(x) einer reellen Zahl x und der Zahl selbst lässt sich mit dem relativen Rundungsfehler ε so darstellen:

$\text{rd}(x) = x(1+\varepsilon)$ oder $(\text{rd}(x) - x)/x = \varepsilon$.

Der relative Rundungsfehler normalisierter Zahlen ist durch $|\varepsilon| \le \text{eps}$ beschränkt. Für $|x| < \text{min}$, entsteht ein Unterlauf-Fehler (Underflow Error) und es gilt $|\text{rd}(x)-x| \le \text{min}$. Reelle Zahlen, die eine maschineninterne Darstellung ungleich null gestatten, nenne ich einfachheitshalber „deutlich von null verschieden“.

Die Abschätzung des Rundungsfehlers durch eps ist etwas grob. Bei der üblichen Rundung ist der relative Rundungsfehler maximal gleich dem *halben* Abstand benachbarter Gleitpunktzahlen.

Die Formeln der Fehlerabschätzung werden in den Programmen genau anders herum verwendet: Wir kennen die Maschinenzahl und versuchen daraus eine Abschätzung der reellen Zahl zu gewinnen. Das heißt: die Formel $\text{rd}(x) = x(1+\varepsilon)$ ist nach x aufzulösen. Unter Vernachlässigung der Produkte von Epsilonwerten gilt $x = \text{rd}(x) \cdot (1-\varepsilon)$. Der relative Rundungsfehler ist durch dieselbe Schranke abschätzbar wie vorher. Das heißt: Wir können in die Formeln der

Fehlerabschätzung im Allgemeinen anstelle der - meist nicht bekannten reellen Zahlen - die Gleitpunktnäherungen verwenden. Dass wir Produkte von Epsilonwerten vernachlässigen, ist nicht weiter schlimm. Durch die Wahl eines geringfügig größeren Wertes für eps werden diese Terme mit abgedeckt.

Fehlerfortpflanzung

Die Maschinenoperationen wollen wir uns so vorstellen, dass das Ergebnis gleich dem gerundeten Ergebnis der exakten Rechenoperation ◊ ist: rd(rd(x_1) ◊ rd(x_2)) ist also das maschineninterne Resultat der Verknüpfung der Gleitpunktdarstellungen der reellen Zahlen x_1 und x_2. Mit den durch eps absolut begrenzten Zahlen ε_1, ε_2, und ε_3 gilt:

$$\mathrm{rd}(\mathrm{rd}(x_1) \lozenge \mathrm{rd}(x_2)) = (x_1(1+\varepsilon_1) \lozenge x_2(1+\varepsilon_2))(1+\varepsilon_3)$$

Konkret heißt das für die Addition:

$$\begin{aligned}\mathrm{rd}(\mathrm{rd}(x_1) + \mathrm{rd}(x_2)) &= (x_1(1+\varepsilon_1) + x_2(1+\varepsilon_2))(1+\varepsilon_3)\\ &= (x_1+x_2)(1 + \frac{x_1}{x_1+x_2}\,\varepsilon_1 + \frac{x_2}{x_1+x_2}\,\varepsilon_2)(1+\varepsilon_3)\\ &= (x_1+x_2)(1 + \frac{x_1}{x_1+x_2}\,\varepsilon_1 + \frac{x_2}{x_1+x_2}\,\varepsilon_2 + \varepsilon_3 + \ldots)\\ &= (x_1+x_2)(1+\varepsilon)\end{aligned}$$

Die Pünktchen repräsentieren Terme, in denen Produkte von Epsilonwerten vorkommen. Diese Terme sind folglich so klein, dass sie keine Rolle spielen. Durch eine geringfügige Vergrößerung der Konstanten eps können sie in den Abschätzungen berücksichtigt werden. Der relative Fehler des Ergebnisses lässt bei Vernachlässigung von Epsilonprodukten so abschätzen:

$$|\varepsilon| \le (\frac{|x_1|+|x_2|}{|x_1+x_2|}+1)\cdot\mathrm{eps} \qquad \text{(Fehlerabschätzung bei Addition)}$$

Falls die Summanden gleiches Vorzeichen haben, vereinfacht sich die Formel zu $|\varepsilon| \le 2\cdot$eps. Gefährlich ist der andere Fall, dass die Zahlen verschiedene Vorzeichen haben. Wenn sie dann auch noch betragsmäßig nahezu gleich groß sind, kommt es zur *Stellenauslöschung*.

Beispiel: Bei Addition der auf vier Nachpunktstellen gerundeten Zahlen 1.2345 und -1.2344 entsteht das Ergebnis 0.0001. Nehmen wir einmal an, die erste der Zahlen ist die Maschinendarstellung (hier ausnahmsweise zur Basis 10) von x_1 = 1.23445 und die zweite der Zahlen ist das Rundungsergebnis der Zahl x_2 = -1.2344499... mit unendlich vielen Neunen nach dem Dezimalpunkt. Offenbar sind die Zahlen – abgesehen vom Vorzeichen – identisch. Ihre Summe ist gleich null. Dem berechneten Ergebnis kann man das nicht mehr ansehen. Vor allem nützt es gar nichts, wenn man den Rundungsfehler mit der obigen Formel abschätzen will und – mangels Originalwerten – die jeweils entsprechenden Gleitpunktzahlen einsetzt. Im Beispiel kann man eps = 0.00005 setzen. Die Fehlerabschätzung würde bei Ersetzung der reellen Zahlen durch ihre Gleitpunkt-Näherung eine Obergrenze von ((1.2345 + 1.2344)/0.0001+1)·eps = 24690·0.00005 = 1.2345 für den relativen Rundungsfehler ergeben. Dem Wert sieht man zwar schon an, dass das Ergebnis nichts taugt. Tatsächliche aber ist der relative Rundungsfehler unendlich groß. Bei der Rechenoperation selbst entsteht paradoxerweise überhaupt kein Rundungsfehler (ε_3 = 0).

Gegenüber Addition und Subtraktion sind Multiplikation und Division harmlos:

$$\mathrm{rd}(\mathrm{rd}(x_1) \cdot \mathrm{rd}(x_2)) = (x_1(1+\varepsilon_1) \cdot x_2(1+\varepsilon_2))(1+\varepsilon_3)$$

$$= x_1 \cdot x_2(1+\varepsilon_1)(1+\varepsilon_2)(1+\varepsilon_3)$$

$$= x_1 \cdot x_2(1+\varepsilon_1+\varepsilon_2+\varepsilon_3 + ...)$$

$$= x_1 \cdot x_2(1+\varepsilon)$$

Der relative Rundungsfehler ε lässt sich also durch 3·eps abschätzen. Dasselbe gilt für die Division.

$|\varepsilon| \leq 3 \cdot \text{eps}$ (Fehlerabschätzung bei Multiplikation und Division)

Die Formeln der Fehlerabschätzung behandeln den allgemeinen Fall, dass die Operanden selbst Rundungsergebnis sind.

Soweit gelten die Formeln für die rechnerinterne Verknüpfung von möglichst genau repräsentierten Zahlen: Die Operanden wurden als jeweils mit nur einem einfachen Rundungsfehler behaftet vorausgesetzt. Aber auch der Fehlerfortpflanzung kommt man mit derselben Methode bei.

Von einem dramatischen Fall von Fehlerfortpflanzung wird in „Problem Numerik: Rundungs- und Verfahrensfehler“ berichtet (www.fh-fulda.de/~grams/SimMaterial/ProblemNumerik).

Literaturhinweise: Stoer (1994, S. 5 ff), Hammer/Hocks/Kulisch/Ratz (1995, S. 42 ff.), Schwetlick/Kretzschmar (1991, S. 17).

Aufgaben

9.1 Beweisen Sie, dass aus $\text{rd}(x) = x(1+\varepsilon)$ näherungsweise $x = \text{rd}(x)\cdot(1-\varepsilon)$ folgt.

9.2 Vergleichen Sie Ihre Lösung der Aufgabe 1.5 mit der korrekten Lösung $\pi^2/6$. Was ist schief gelaufen? Wie lässt sich die Sache verbessern? Schreiben Sie ein Programm, das in möglichst kurzer Zeit das Ergebnis im double-Format mit höchstmöglicher Genauigkeit errechnet.

Hinweis: Ändern Sie die Reihenfolge der Summation und nutzen Sie folgende Formel für die Abschätzung des Reihenrests: $\sum_{i=n}^{m} \frac{1}{i^2} \leq \sum_{i=n}^{m} \frac{1}{i(i-1)} = \sum_{i=n}^{m} \frac{1}{i-1} - \frac{1}{i} = \frac{1}{n-1} - \frac{1}{m}$.

Leiten Sie eine analoge Formel für die Untergrenze her. Beziehen Sie die Schätzung des Reihenrestes in die Ergebnisdarstellung ein!

9.3 Die Berechnung von Mittelwert m und Standardabweichung s einer Stichprobe hat man die Formeln

$$m = (\sum_{i=1}^{N} x_i) / N \text{ und } s = \sqrt{\frac{1}{N-1} \cdot \sum_{i=1}^{N} (x_i - m)^2}$$

Die Formel für die Standardabweichung hat einen entscheidenden Nachteil: Alle Stichprobenwerte müssen für die Berechnung der Standardabweichung aufgehoben werden. In vielen Fällen ist die folgende Formel günstiger:

$$s = \sqrt{\frac{1}{N-1} \cdot (\sum_{i=1}^{N} x_i^{\,2} - N m^2)}$$

Hier müssen die Stichprobenwerte nur linear und quadratisch akkumuliert werden. Die Werte selbst werden für die letztendliche Ermittlung von m und s nicht mehr benötigt. Aber dafür hat man sich einen anderen Nachteil eingehandelt: Das Programm endet ab und zu mit einer Fehlermeldung. Woran liegt das? Probieren Sie die Sache im float-Format aus und wählen Sie

Stichprobenwerte, die im Intervall [0, $_{10}$-23) gleichverteilt sind. Wie man die numerische Stabilität der ersten Formel mit den Vorteilen der zweiten vereinbart, ist im Buch von Knuth (II, 1981, S. 216) zu finden.

9.4* Gesucht ist eine Näherungslösung des Anfangswertproblems

$$\dot{x} = \frac{9}{10}x(x-1) + xy$$

$$\dot{y} = y(y-1) + \frac{9}{10}xy$$

$$x(0) = y(0) = \frac{1}{2}$$

für das Zeitintervall [0, 100] mittels numerischer Integration. Diskutieren Sie das Ergebnis und führen Sie eine Fehleranalyse durch.

Hinweis: Führen Sie als Kontrollvariable die Summe z = x + y ein. Bestimmen Sie deren Verlauf auf analytischem Weg.

10 Intervallarithmetik und Fixpunkte

Bezeichnungen: Für die Darstellung der Grundideen der *Intervallarithmetik* sind ein paar Definitionen nützlich:

$[x]$ Kurzbezeichnung eines reellen Intervalls $[x_l, x_u]$

$d([x])$ Durchmesser des Intervalls $[x]$: $d([x]) = x_u - x_l$

$m([x])$ Mittelpunkt des Intervalls $[x]$: $m([x]) = (x_l + x_u)/2$

$\boldsymbol{R}$ Menge der reellen Zahlen

$\boldsymbol{IR}$ Menge der reellen abgeschlossenen Intervalle $[a, b]$ mit $a, b \in \boldsymbol{R}$

$\boldsymbol{M}$ Menge der *Gleitpunkt-Zahlen*, die im Computer tatsächlich darstellbar sind (Floating-Point Numbers, *Maschinenzahlen*)

$\boldsymbol{IM}$ Menge der Maschinenintervalle - das sind reelle Intervalle $[a, b]$ mit $a, b \in \boldsymbol{M}$. Wichtig: Ein Intervall aus $\boldsymbol{IM}$ enthält im Allgemeinen nicht nur Element aus $\boldsymbol{M}$!

Frage: Wann sind sämtliche Elemente eines Maschinenintervalls Maschinenzahlen?

Alle Berechnungen spielen sich im Bereich der Maschinenzahlen ab. Die numerische Behandlung eines mathematischen Problems kann also bestenfalls eine Approximation $\tilde{x} \in \boldsymbol{M}$ an die korrekte Lösung $x \in \boldsymbol{R}$ liefern. Sind zur Ermittlung des Ergebnisses mehrere Rechenschritte erforderlich, werden sich die Abweichungen vom korrekten Ergebnis fortpflanzen und mit weiteren Rundungsfehlern überlagern.

Reelle Intervallarithmetik: Die Intervallrechnung arbeitet von vornherein mit Intervallen, die die korrekten Daten enthalten. Beispielsweise ist [3.1415, 3.1416] ein Intervall, das die Kreiszahl π enthält. Je kleiner der Durchmesser eines Intervalles ist, umso genauer wird das betreffende Datum dargestellt. Wenn der Durchmesser gleich null ist, enthält das Intervall nur einen Wert und das Intervall ist nur eine andere Darstellung der Zahl selbst.

Alle mathematischen Operationen werden auf Intervalle ausgedehnt. Das geschieht so, dass das Ergebnisintervall einer Rechenoperation das kleinstmögliche Intervall ist, das alle möglichen Ergebnisse enthält, wenn man irgendeine Zahl aus dem ersten mit irgendeiner Zahl aus dem zweiten der Intervalle wählt und diese mit der entsprechenden Rechenoperation miteinander verknüpft. Sei $\Diamond$ eine solche Operation ($\Diamond$ steht also für $\cdot$, +, - oder /), dann ist $[x] \Diamond [y]$ das kleinste Intervall, das die Menge $\{\xi \Diamond \eta \mid \xi \in [x], \eta \in [y]\}$ umfasst.

Nach dem eben Gesagten, ist die Intervallarithmetik eine direkte Verallgemeinerung der Zahlenrechnung. Für die Addition erhält man beispielsweise

$$[x]+[y] = [x_l, x_u] + [y_l, y_u] = [x_l + y_l, x_u + y_u].$$

Nun lassen sich Algorithmen direkt in die Intervallarithmetik übersetzen, indem man jede Größe durch ein Intervall und jede Rechenoperation durch die entsprechende *Intervalloperation* ersetzt. Diese Intervallalgorithmen liefern als Resultate Intervalle, die die gesuchten exakten Resultate umfassen.

Intervallarithmetik mit Maschinenzahlen: Wir haben bei der Berechnung der Intervallgrenzen so getan, als sei das Ergebnis der Rechenoperation im Sinne der reellen Zahlenrechnung korrekt. Das stimmt aber nicht: Auch hier kommt es zu Rundungsfehlern. Die Intervallgrenzen lassen sich also stets nur bis auf den Rundungsfehler genau bestimmen.

Bei der Bestimmung der Intervallgrenzen des Ergebnisses einer Intervalloperation wollen wir die Ergebnisse der Gleitpunktrechnung so runden, dass das tatsächliche im errechneten Intervall enthalten ist. Die Untergrenze wird also nötigenfalls verkleinert und die Obergrenze ver-

größert. Die Formeln für die Fehlerabschätzung vereinfachen sich erheblich, da die Operanden x_1 und x_2 selbst Maschinenzahlen sind: Zwischen dem errechneten Resultat $\mathrm{rd}(x_1 \lozenge x_2)$ und dem wahren Wert besteht die Beziehung

$$\mathrm{rd}(x_1 \lozenge x_2) = (x_1 \lozenge x_2)(1+\varepsilon)$$

mit dem durch eps absolut begrenzten relativen Fehler $|\varepsilon| \le \mathrm{eps}$.

Intervall-Erweiterung: Sei $[a, b]$ ein Intervall mit den frisch errechneten Maschinenzahlen a und b. Um das tatsächliche Intervall abzudecken, wird dieses Intervall geringfügig vergrößert. Die Untergrenze a wird durch $\min\{a(1\text{-eps}), a(1+\mathrm{eps})\}$, die Obergrenze b wird durch $\max\{b(1\text{-eps}), b(1+\mathrm{eps})\}$ ersetzt.

Auf diese Weise enthält man grundsätzlich pessimistische Fehlerabschätzungen. Dem statistischen Hin- und Her der Rundungsfehler trägt die Methode nicht Rechnung. Wenn es nur darum geht, das Ergebnis des Produktes zweier sehr großer Matrizen abzuschätzen, wird man recht große Ergebnisintervalle erhalten.

Anders ist das bei Algorithmen, die auf dem Fixpunktsatz beruhen. Hier führt der Algorithmus im Allgemeinen zu einer zunehmenden Reduktion der Intervallgröße und das Ergebnis ist erreicht, wenn der Algorithmus keine Reduktion des Intervalls mehr erzielen kann.

Bei den mathematischen *Bibliotheks-Funktionen* setzen wir höchstmögliche Genauigkeit voraus. Wir nehmen an, dass sie stets ein Ergebnis liefern, das aus dem exakten Ergebnis durch Rundung entsteht. Also zu Beispiel $\exp(x) = \mathrm{rd}(e^{\mathrm{rd}(x)})$.

Fixpunkt-Bestimmung durch Intervall-Kontraktion

Wir wenden uns wieder der Bestimmung des Fixpunkts der Funktion $f(x) = e^{-x}$ zu. Da die Funktion monoton fallend ist, lassen sich die Bilder der Intervalle leicht ermitteln:

$$f([x]) = [f(x_u),\ f(x_l)].$$

Angenommen, das ursprüngliche Intervall $[x]$ enthält den Fixpunkt ξ. Der Mittelwertsatz der Differentialrechnung garantiert die Existenz zweier Zahlen α und β aus $[x]$ derart, dass

$$f(x_l) = f(\xi) + (x_l - \xi)\, f'(\alpha) = \xi + (x_l - \xi)\, f'(\alpha)$$

$$f(x_u) = f(\xi) + (x_u - \xi)\, f'(\beta) = \xi + (x_u - \xi)\, f'(\beta)$$

Da im Intervall die Ableitung der Funktion stets dasselbe (nämlich ein negatives) Vorzeichen hat, und da das ursprüngliche Intervall den Fixpunkt enthält, haben die Werte $(x_l - \xi)\, f'(\alpha)$ und $(x_u - \xi)\, f'(\beta)$ verschiedene Vorzeichen. Also gilt $\xi \in f([x])$. Das neue Intervall enthält den Fixpunkt also ebenfalls und es ist gegenüber dem ursprünglichen um wenigstens den Faktor L geschrumpft:

$$\mathrm{d}(f([x])) = f(x_l) - f(x_u) = (x_l - \xi)\, f'(\alpha) - (x_u - \xi)\, f'(\beta) \le |x_l - \xi|\, L + |x_u - \xi| \cdot L = \mathrm{d}([x]) \cdot L.$$

Das Intervall $f([x])$ ist kleiner als $[x]$. Dennoch muss es nicht unbedingt ganz in $[x]$ enthalten sein. Da beide Intervalle den Fixpunkt enthalten, kann man als neues Ausgangsintervall die Schnittmenge der beiden bilden. Damit kommen wir zu folgendem *Iterationsschema für die Fixpunktbestimmung mittels Intervallrechnung:*

$$[x_{k+1}] = f([x_k]) \cap [x_k] \text{ für } k = 0, 1, 2, \ldots \qquad \text{(Fixpunkt-Iteration der Intervallrechnung)}$$

wobei $[x_0]$ das Anfangsintervall ist. In unserem Beispiel ist $[x_0] = [½, 1]$.

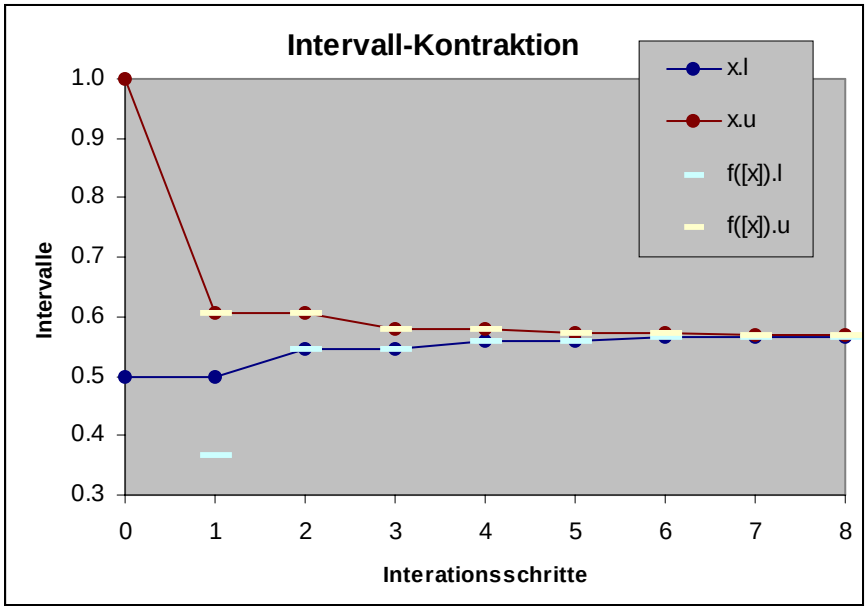
Bisher wurde reelle Intervallrechnung angewendet. Die Übertragung in ein Programm mit Gleitpunktarithmetik erfordert, dass die Intervalle bei jedem Schritt gegebenenfalls leicht vergrößert werden, so dass sie die korrekten Intervalle jedenfalls einschließen. Das geschieht mit

der Intervall-Erweiterung. Die Iteration wird beendet, wenn sich das Intervall nicht mehr verringern lässt. Hier werden tatsächlich Gleitpunkt-Zahlen auf Gleichheit abgefragt. Und das ist völlig ungefährlich. Es folgen das Programm und eine Grafik des Ergebnisses.

```
const float eps= FLT_EPSILON;

typedef struct {float l, u;} interval;

interval f(interval x) {
  interval y;
  y.l=exp(-x.u)*(1-eps);
  y.u=exp(-x.l)*(1+eps);
  return y;
}

interval intersect(interval x, interval y) {
  if (y.l<x.l) y.l=x.l;
  if (x.u<y.u) y.u= x.u;
  return y;
}

void main() {
  interval x={0.5, 1}, y;

  printf("\nFixpunkt von f(x)=exp(-x), Timm Grams, Fulda, 16.01.02");

  do {
    x=f(y=x);
    x=intersect(x, y);
    printf("\n! x.l = %.20g x.u= %.20g", x.l, x.u);
  } while(x.l!=y.l||x.u!=y.u);
}
```

Aufgaben

10.1 Berechnen Sie von Hand mittels Intervall-Kontraktion den Fixpunkt der Funktion $f(x)=1-x/2$.



10.2 Ermitteln Sie mittels Intervalliteration näherungsweise den Wert x, für den $\cos x = x$ gilt. Wenden Sie das Verfahren der Fixpunkt-Iteration direkt auf die Cosinus-Funktion an. Prüfen Sie die Voraussetzungen und grenzen Sie das Startintervall geeignet ein.

11 Intervall-Version des Newton-Verfahrens

Das Intervall-Newton-Verfahren

Sei $f(x)$ eine stetig differenzierbare Funktion mit einer Nullstelle ξ im Intervall $[a, b]$. Diese Nullstelle ist gesucht. Da für Fixpunkte ein passendes Iterationsverfahren existiert, kann man das Problem entsprechend umformulieren: Die Nullstelle der Funktion f ist Fixpunkt der Funktion g, die definiert ist durch $g(x) = x+h(x)\,f(x)$, wobei wir in der Wahl von h noch frei sind, nur verschwinden sollte sie im Fixpunkt nicht. Wir verlangen also $h(\xi) \neq 0$. Wir haben gesehen, dass die Fixpunkt-Iteration umso besser funktioniert, je kleiner der Absolutwert der Ableitung der Funktion ist. Wenn wir verlangen, dass $g'(\xi) = 0$ sein soll, ergibt sich $1+h'(\xi)\,f(\xi) +h(\xi)\,f'(\xi) = 1+ h(\xi)\,f'(\xi) = 0$, also: $h(\xi) =-1/f'(\xi)$. Offenbar ist $-1/f'(x)$ eine gute Wahl für $h(x)$.

Die Nullstellensuche für $f(x)$ läuft also auf die Fixpunktsuche für die Funktion

$$g(x) = x - f(x)/f'(x)$$

hinaus. Unter bestimmten Bedingungen mit geeignetem Anfangswert x_0 konvergiert die durch $x_{k+1} = g(x_k)$ definierte Folge gegen die Nullstelle. Das funktioniert jedenfalls dann, wenn die Funktion $g(x)$ auf dem Intervall $[a, b]$ eine durch eine Konstante L absolut beschränkte Ableitung besitzt, wenn diese Konstante kleiner eins ist, und wenn $x_0 \in [a, b]$. Das ist das *Iterationsverfahren von Newton* zur Nullstellenbestimmung.

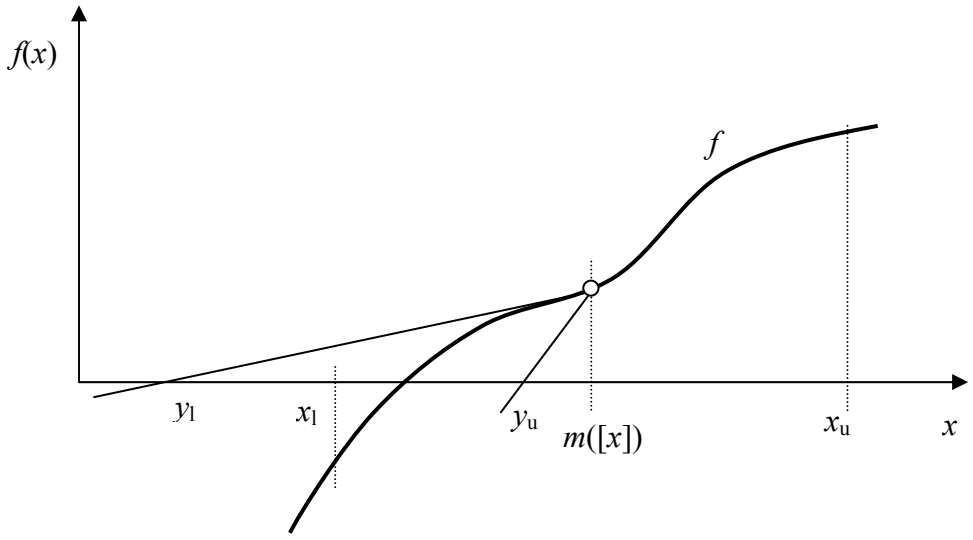


Bild 11.1 Schritt des Intervall-Newton-Verfahrens

Man könnte nun versuchen, auf g das Verfahren der Intervall-Kontraktion des vorigen Abschnitts anzuwenden. Aber es geht geschickter.

Möge ein Intervall $[a, b]$ eine Nullstelle ξ der Funktion f enthalten. Es wird also angenommen, dass $\xi \in [a, b]$ und $f(\xi) = 0$. Auf dem Intervall möge die Ableitung f' der Funktion existieren, stetig und ungleich null sein. Für einen beliebig gewählten Punkt $x \in [a, b]$ existiert - wegen des Mittelwertsatzes - ein Punkt $\eta \in [a, b]$, so dass

$$f(x) = f(\xi)+ (x - \xi)\,f'(\eta) = (x - \xi)\,f'(\eta)$$

Daraus folgt

$$\xi = x - f(x)/ f'(\eta)$$

Nun setzen wir an Stelle von η das Intervall $[a, b]$ ein und erhalten

$\xi \in x - f(x)/f'([a, b])$

Das heißt, die Nullstelle ist auch in dem Intervall $x - f(x)/f'([a, b])$ enthalten.

Um diese Überlegungen iterativ weiterführen zu können, ersetzen wir das Intervall $[a, b]$ durch ein Intervall $[x]$ und den Punkt x durch den Mittelpunkt des Intervalls $m([x])$. Damit definieren wir die Intervall-Funktion N durch

$N([x]) = m([x]) - f(m([x]))/f'([x])$ (Intervallfunktion des Newton-Verfahrens)

Die so definierte Intervall-Funktion N hat also die schöne Eigenschaft, dass jede Nullstelle $\xi \in [x]$ auch im neu berechneten Intervall liegt: $\xi \in N([x])$.

Die Intervallauswertung der Ableitung f' ist der heikle Part der Rechnung: Wenn f' im Intervall $[a, b]$ keine monotone Funktion ist, und man auch sonst wenig von ihr weiß, wird es schwierig.

Bild 11.1 veranschaulicht die Intervall-Funktion: Mit $[y]$ wird das Bild des Intervalls $[x]$ bezeichnet: $[y] = N([x])$. Der Funktionswert an der Stelle $m([x])$ ist mit einem Kreis markiert. Wir gehen o.B.d.A davon aus, dass sowohl $f(m([x]))$ als auch die Steigung der Funktion f positiv sind. Die Minimal und Maximalwerte von f' auf dem Intervall $[x]$ bezeichnen wir mit f'_l und f'_u. Offenbar gilt unter den gemachten Annahmen, dass $y_l = m([x]) - f(m([x])/f'_l$, und das ist gleichbedeutend mit $f(m([x]))/(m([x]) - y_l) = f'_l$. Wir sehen: f'_l ist die Steigung der vom markierten Punkt nach links unten weggehenden Geraden. Analog ergibt sich, dass f'_u die Steigung der nach rechts weggehenden Geraden ist.

Die *Intervall-Iteration* wird definiert durch

$[x_{k+1}] = N([x_k]) \cap [x_k]$ für $k = 0, 1, 2, ...$ (Intervall-Newton-Verfahren)

Wenn die Intervallfolge mit dem Anfangswert $[x_0] = [a, b]$ kontrahierend ist, haben wir damit eine Näherungslösung für unser Problem (Hammer/Hocks/Kulisch/Ratz, 1995, S. 93 ff.).

Aufgaben

11.1 Leiten Sie die Wertetabellen für die Intervall-Multiplikation und die Intervall-Division her.

$[x]\cdot[y]$	$0\le y_l$	$y_l<0<y_u$	$y_u\le 0$
$0\le x_l$			
$x_l<0<x_u$			
$x_u\le 0$			

$[x]/[y]$	$0\le y_l$	$y_u\le 0$
$0\le x_l$		
$x_l<0<x_u$		
$x_u\le 0$		

11.2 Schreiben Sie ein allgemein verwendbares Modul zur Bestimmung der Nullstelle einer Funktion mittels Intervall-Newton-Verfahren. Setzen Sie voraus, dass die Ableitung der

Funktion auf dem Startintervall $[x_0]$ monoton ist. Machen Sie für alle Werte Gleitpunktabschätzungen.

11.3 Ermitteln Sie mittels Intervalliteration näherungsweise den Wert x, für den $\cos x = x$ gilt. Wenden Sie das Intervall-Newton-Verfahren auf die durch $f(x) = x - \cos x$ definierte Funktion f an. Prüfen Sie die Voraussetzungen und grenzen Sie das Startintervall geeignet ein. Vergleichen Sie das Ergebnis mit dem aus Aufgabe 10.2. Was gibt es zur Effizienz der beiden Verfahren zu sagen.

11.4 Schreiben Sie ein Programm zur Berechnung der n-ten Wurzel einer Zahl. Verwenden Sie das Intervall-Newton-Verfahren. Schreiben Sie ein weiteres Programm auf der Basis der Intervallteilung (Bisektion). Vergleichen Sie die Ergebnisse und machen Sie Aussagen zur Genauigkeit.

12 Programmierstile

Am Beispiel eines einfachen Automaten werden grundlegende Programmierstile besprochen. Der Abschnitt soll auf weiterführende Lehrveranstaltungen einstimmen: Programmkonstruktion (mit Java) und Simulation.

Aufgabe: Verkaufsautomat

Automat: Erstellen Sie das Simulationsprogramm für einen einfachen Verkaufsautomaten (Böttcher/Kneißl, S. 162 ff.). Er hat die folgenden Eingabemöglichkeiten: **M**ünzeinwurf, Rückknopf **D**rücken, Rückknopf **L**oslassen, Schublade **A**uf und Schublade **Z**u. Die fett gedruckten Buchstaben nehmen wir zur Abkürzung der Aktionen: M, D, L, A und Z. Die Reaktionen des Automaten bestehen darin, gewisse Riegel auf- oder zuzumachen: **S**chub **a**uf, **S**chub **z**u, **A**uswurf **a**uf, **A**uswurf **z**u, **K**asse **a**uf und **K**asse **z**u. Dafür nehmen wir die Abkürzungen Sa, Sz, Aa, Az, Ka und Kz. Falls in einem Schritt nichts ausgegeben wird, bezeichnen wir das als NoOp (**No Op**eration). Wir gehen davon aus, dass der Programmentwurf bereits durchgeführt worden ist. Das Entwurfsergebnis, der Zustandsübergangsgraph eines Mealy-Automaten (Bild 12.1), ist Basis der Programmentwicklung.

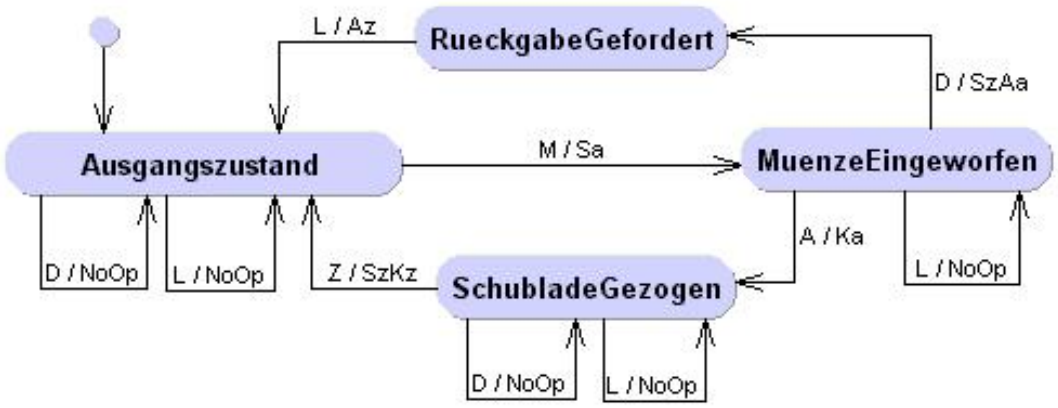


Bild 12.1 Zustandsübergangsgraph des Verkaufsautomaten (erstellt mit dem UML-Werkzeug GO)

Lösungansätze

Assemblerstil

Beim Assemblerstil identifizieren wir jeden der Zustände mit einem Programmabschnitt, der auf ein entsprechendes Label folgt. Als Beispiel wird der Programmabschnitt des Ausgangszustands dargestellt

```
Ausgangszustand:
  e=toupper(getch());
  switch(e){
    case 'M': printf("Sa\n");  goto MuenzeEingeworfen;
    case 'L': printf("NoOp\n"); goto Ausgangszustand;
    case 'D': printf("NoOp\n"); goto Ausgangszustand;
    case '.': goto Stop;
    default: printf("Eingabefehler\n"); goto Ausgangszustand;
  }
```

Objektorientierter Stil

Jeder Zustand wird durch ein Objekt des Typs `Zustand` repräsentiert:

```
#define Zustand struct Zustand
```

```
Zustand {Zustand *(*Transition)(char e);}
  Ausgangszustand, MuenzeEingeworfen,
  RueckgabeGefordert, SchubladeGezogen;
```

Die Objekte werde hier gleich auf dem Stack angelegt. Es ist es nicht erforderlich, den Objekten Membervariable mitzugeben, da alle Reaktionsmöglichkeiten der Objekte mit Hilfe der Funktionsvariablen `Transition` realisiert werden. Für den Ausgangszustand sieht die Realisierung dieser der Funktion `Transition` so aus:

```
Zustand *A(char e){
 if(e=='M') {printf("Sa\n");  return &MuenzeEingeworfen;}
 else if (e=='L' || e=='D') printf("NoOp\n");
 else printf("Eingabefehler\n");
 return &Ausgangszustand;
}
```

Die Steuerung des Programmablaufs mittels der Eingabevariablen `e` geschieht durch die Anweisung `z=z->Transition(e);`

Reduktion des Objekts auf die Funktion

Dieser Lösungsvorschlag entsteht aus dem objektorientierten Ansatz durch Verzicht auf die einhüllende Struktur. Der Zustand wird durch die Funktionsvariable selbst repräsentiert:

```
typedef void *(*Zustand)(char);
```

Die Steuerung des Programmablaufs sieht jetzt so aus:

```
z=z(e);
```

Dabei bezeichnet `z` den aktuellen Zustand.

Explizite Zustandscodierung

Bei der expliziten Zustandscodierung wird der Zustand durch eine Zahl repräsentiert.

```
#define Zustand enum Zustand
Zustand {Ausgangszustand, MuenzeEingeworfen,
         RueckgabeGefordert, SchubladeGezogen};
```

Ablaufsteuerung und zustandabhängige Reaktionen werden mittels einer in eine Schleife eingebetteten Switch-Anweisung realisiert. Jedem Zustand ist eine Case-Anweisung zugeordnet.

Tabellenbasierter Stil

Die Zustände selbst werden wie bei der expliziten Zustandcodierung durch Integer-Werte repräsentiert. Ergänzend wird ein Zustand `Stop` als Sentinel eingeführt. Der Zustandsgraph wird in einem Assoziativarray abgelegt. Diese Tabelle legt man am besten statisch an und initialisiert sie gleich bei der Deklaration.

```
struct {char Zustand, Bedingung, *Aktion, Ziel;} Zustand[]=
  {{Ausgangszustand,   'M', "Sa\n",   MuenzeEingeworfen},
   {Ausgangszustand,   'L', "NoOp\n", Ausgangszustand},
   ...
   {Stop,              0,   0,        0}
  };
```

Für den Ablauf wird die Tabelle nach dem aktuellen `Zustand` und der passenden `Bedingung` (=Eingabe) durchsucht. Falls gefunden, wird die Aktion durchgeführt und dem aktuellen Zustand der Wert `Ziel` zugewiesen.

Verteilte Tabellen

Bei dieser Lösung wird die Tabelle auf die Zustände aufgeteilt. Auf diese Weise wird die flache Suchstruktur der vorhergehenden Lösung vermieden. Datenstruktur:

```
typedef
  struct Trans {char Bedingung, *Aktion; struct Trans *Ziel;} Transition;

extern Transition
  MuenzeEingeworfen[], RueckgabeGefordert[], SchubladeGezogen[];

Transition
  Ausgangszustand[]=    {{'M', "Sa\n",   MuenzeEingeworfen},
                         {'L', "NoOp\n", Ausgangszustand},
                         {'D', "NoOp\n", Ausgangszustand},
                         {0}
                        },
                        ...
  RueckgabeGefordert[]= {{'L', "Az\n",   Ausgangszustand},
                         {0}
                        },
  SchubladeGezogen[]=   {{'Z', "SzKz\n", Ausgangszustand},
                         {'L', "NoOp\n", SchubladeGezogen},
                         {'D', "NoOp\n", SchubladeGezogen},
                         {0}
                        };
```

Diskussion der Lösungsvorschläge

Der *Assembler-Stil* Lösung ist der Sache angemessen. Die Lösung ist übersichtlich. Die Verwendung des geächteten `goto` lässt sich hier ausnahmsweise rechtfertigen: Die Struktur bleibt transparent. Die *Redundanz des Programmtexts* ist etwas unangenehm: In jedem Programmabschnitt müssen gewisse Eingabe- und Ausgabefunktionen wiederholt werden. Wesentliches Manko: Die Ablaufsteuerung ist über das gesamte Programm verstreut.

Bei der *objektorientierten* Lösung werden die Zustände zu Objekten gemacht. Das räumt mit gewissen Redundanzen auf. Die Lösung lässt sich leicht auf wesentlich kompliziertere Fälle übertragen. Die Anwendung dieser Programmiertechnik wirkt hier allerdings etwas überzogen: Die - bei dieser Programmtechnik prinzipiell mögliche - *Kapselung von Daten und Funktionen* kommt nicht zum Tragen, da die Zustände ohne Daten auskommen. Die *Ablaufsteuerung* lässt sich äußerst prägnant formulieren: `z=z->Transition(e);`

Die *funktionsorientierte* Lösung lässt von den Objekten nur die Funktionen übrig. Sie ist eine konsequente Vereinfachung der objektorientierten Lösung. Die Eleganz der Ablaufsteuerung ist gerettet: `z=z(e);`

Bei der *expliziten Integer-Zustandscodierung* erkauft man die Transparenz der Lösung und die Bereinigung der Redundanzen mit einer *flachen und breiten Suchstruktur:* In jedem Schritt werden die Zustände *und* die Eingaben abgefragt. Weiteres Manko: Wie beim Assembler-Stil ist die Ablaufsteuerung über das Programm verstreut.

Die *Tabellen-Lösung* ist sehr übersichtlich und vermeidet einige Redundanzen. Die Ablaufsteuerung ist konzentriert. Aber wie bei der expliziten Integer-Zustandscodierung ist die Suchstruktur flach.

Die Lösung mit den *verteilten Tabellen* vermeidet den Nachteil der Tabellenlösung: Die Suche beschränkt sich auf die Möglichkeiten innerhalb des aktuellen Zustands. Die Ablaufsteuerung ist vom Automatenprogramm fein säuberlich getrennt. Die Effizienz ist gewahrt und Redundanz weitgehend vermieden.

Sachverzeichnis

N

O

P

Q

R

S

T

U

V

W

Z