Projektgruppe Praktische Mathematik
Technische Universität Berlin
Sekretariat MA 4-3

Einführung in die Programmiersprache

Fortran 95

Projektgruppe Praktische Mathematik *

19. Juni 2008

*Quellen: ~ppm-soft/cvsroot/skripte/fortran/
Zuletzt bearbeitet: Gert Herold (19. Juni 2008)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Computer nehmen in unserem Leben immer mehr Raum ein.

Informationsgesellschaft, Datenautobahn, Internet, MultiMedia sind nur einige Stichworte, die in der westlichen Industriegesellschaft immer mehr Menschen das Gefühl geben, dass auch sie einen Computer in ihren eigenen vier Wänden brauchen.

Die meisten Benutzer dieser Geräte betrachten ihren Computer mehr oder weniger ehrfürchtig wie eine Zauberkiste. Umfangreiche Softwarepakete vermitteln ihnen das Gefühl, dass das Gerät intelligent sei, sogar intelligenter als sie selber, denn: kann es nicht lauter Dinge, die sie nicht können?

Der Blick auf die Tatsache, dass ein Computer nur das kann, was ihm einprogrammiert wurde und dieses nur so gut oder schlecht, wie der Programmierer dazu in der Lage war, wird im Normalfall durch mehr oder weniger bunt schillernde Oberflächen versperrt.

Wenn man grundlegend begriffen hat, wie ein Computer funktioniert und Programme arbeiten, verblasst der magische Effekt der Rechner mit ihrer Software und ein Werkzeug bleibt übrig. Zu welchen Zwecken es sinnvoll eingesetzt werden kann, muss sich jeder selbst überlegen.

Diese Einführung in die Programmiersprache Fortran95 soll Interessierten die Möglichkeit bieten, selber Programme zu schreiben. Der Schwerpunkt liegt auf den Befehlen, die für mathematisches und naturwissenschaftliches Arbeiten notwendig sind. Hat man die Struktur der Programmiersprache begriffen, sollte es auch möglich sein, sich in andere Problembereiche einzuarbeiten, wenn man dieses braucht.

In diesem Kurs wird viel Wert auf eine strukturierte Programmierung gelegt und in diesem Zusammenhang die Verwendung von Struktogrammen verlangt. Es gibt sicherlich auch andere Programmierphilosophien, aber gerade bei der Programmierung von numerischen Lösungen hat sich diese Methode bewährt und deshalb wird hier nur die strukturierte Verwendung von Fortran95 vorgestellt.

1.1 Wie soll mit diesem Skript gearbeitet werden?

Dieses Skript hat den Anspruch, das weitgehend autodidaktisch mit ihm gearbeitet werden kann:

- Es soll aufmerksam durchgearbeitet werden
- Die Zusammenarbeit in Zweiergruppen ist dringend erwünscht
- Für auftretende Fragen sind Tutoren und Tutorinnen zur Unterstützung da

Dieses Skript kann und soll kein vollständiges Fortran-Nachschlagewerk sein. Es ist deshalb empfehlenswert, für weiterführende Informationen entsprechende Bücher sowie das Internet zu Rate zu ziehen.

1.2 Geschichte der Computer

Dieses Kapitel kann zu Kursbeginn übersprungen werden, und irgendwann in einer ruhigen Minute (Cafeteria, Park ...) gelesen werden.

> Denn es ist eines ausgezeichneten Mannes nicht würdig, wertvolle Stunden wie ein Sklave im Keller der einfachen Rechnungen zu verbringen. Diese Aufgaben könnten ohne Besorgnis abgegeben werden, wenn wir Maschinen hätten.
>
> Gottfried Wilhelm Leibniz, 1646 - 1716.
> Er entwickelte u.a. das binäre Zahlensystem und konstruierte eine Rechenmaschine.

Die Entwicklung von Computern (Rechenmaschinen) ist stark an die Entwicklung der Mathematik und der physikalischen Naturwissenschaften gebunden. Die Entwicklung der Mathematik und der Naturwissenschaften sind wiederum stark an gesellschaftliche Entwicklungen gebunden.

Die Notwendigkeit Zahlen zu entwickeln, entstand mit dem Fortschreiten der Urbanisierung und der Staatengründung. Steuern sollten eingetrieben werden (zunächst noch in Naturalien), große Menschenmengen mussten versorgt werden, logistische Probleme gelöst und Handel getrieben werden. Nur in Kulturen mit Stadt- und Staatsbildung hat sich eine Mathematik und ein größere Mengen beschreibendes Zahlensystem entwickelt.

Schon frühzeitig wurden Hilfsmittel zu der Bewältigung der anfallenden Rechenaufgaben gefunden. Sei es die Entwicklung des Abakus in Ostasien (ca. 1100 v.u.Z.) oder das Rechnen mit Steinen. Von Euklid (ca. 365-300) existiert ein Buch, das bereits zahlreiche Algorithmen, zur Vereinfachung von Rechenvorgängen enthält.

Ein entscheidender Schritt für die Weiterentwicklung der Mathematik war die Einführung der Null, die erst das uns gewohnte Rechnen im Dezimalsystem (und auch im binären Zahlenraum) ermöglicht. Die Null wurde ca. im 7. Jahrhundert unserer Zeitrechnung in Indien eingeführt[1], in Europa erst im 12. - 13. Jahrhundert, mit der Einführung des arabischen Zahlensystems.

Adam Ries (1518-1550), der in der Handelsstadt Nürnberg lebte, entwickelte Rechenbücher zur Formalisierung der dezimalen Rechenmethoden. Im siebzehnten Jahrhundert wurden von verschiedenen Leuten mechanische Rechenmaschinen entworfen (Schickardt, Blaise Pascal, Leibniz u.a.)[2].

Mit der Entwicklung des binären Zahlensystems und seiner Arithmetik legte Leibniz die Grundlagen für die späteren Computerentwicklungen.

Die nächsten entscheidenden Schritte für die Computerentwicklung finden erst im 19. Jahrhundert statt:

Jaquardt entwickelt zu Beginn des Jahrhunderts einen automatischen Webstuhl, der mit Lochkarten gesteuert wird. Charles Babbage – ein englischer Mathematiker, der sich mit Sterbetafeln für Versicherungen (Statistik) und astronomischen Tabellen (im Auftrag des Kriegsministeriums) beschäftigte – baut um 1830 die erste programmgesteuerte Rechenmaschine, die „Analytical engine“. Angeblich hat sie nie so richtig funktioniert. Seine Assistentin, Ada Lovelace, war die erste theoretische Informatikerin.

Um 1855 entwickelt George Boole die nach ihm benannte „**boolesche Algebra**“, die zum Auswerten von logischen Zusammenhängen notwendig ist und auch ein wesentlicher Bestandteil von Computerprogrammen ist.

Ende des Jahrhunderts bekam Hollerith auf einer Bahnfahrt in den USA die Idee, Daten auf Lochkarten zu speichern. Der Gedankenblitz traf ihn, als der Kontrolleur in seine Fahrkarte Löcher knipste, die an verschiedenen Stellen unterschiedliche Bedeutungen hatten (Fahrziel, Preisklasse oder ähnliches). Mit Hilfe der Lochkarten von Hollerith wurde um 1880 eine Volkszählung in den USA durchgeführt. Die Lochkarten wurden maschinell ausgewertet. Diese Erfindung führte zum Einsatz in der Buchhaltung von größeren Unternehmen und zur Betriebsüberwachung (Lagerhaltung, Arbeitszeiten, etc.). Hollerith war Mitbegründer einer Büromaschinenfirma, die sich 1924 in Industrial Business Machines Corporation (kurz: IBM) umbenannte.

Ein technikbegeisterter Bauingenieurstudent der TH Charlottenburg (heute TU Berlin) Namens Konrad Zuse[3], hatte von den immer gleichen Auslegungsberechnungen zur Statik die Nase voll und konstruierte die ersten Rechner in Relaistechnik in der Wohnung seiner Eltern. Den ersten funktionstüchtigen Rechenautomaten in Relaistechnik entwickelte er dann 1941. Zuse war damals bei der Firma Henschel mit

[1] Die Kirche stellte sich eine Zeitlang gegen die Einführung der Null in ihrem Herrschaftsbereich. Das absolute Nichts durfte es anscheinend im Christentum nicht geben - im Buddhismus (der in Indien weit verbreitet war und ist) ist es ein angestrebter Zustand!

[2] Schickardt wollte astronomische Rechnungen automatisieren, Blaise Pascal war der Sohn eines Steuerpächters, den er mit seiner Maschine unterstützen wollte. Die wenigsten der in dieser Zeit entworfenen Maschinen, funktionierten tatsächlich.

[3] Der sich sowohl mit dem „Kapital“ von Marx, als auch mit den Texten von Henry Ford auseinandersetzte.

der Berechnung der Statik von fliegenden Bomben beschäftigt. Seine Erfindung blieb von staatlichen Stellen weitestgehend unbeachtet, wahrscheinlich, weil sie sich nicht zur Verarbeitung großer Datenmengen eigneten, sondern eher für technische Berechnungen.

In dieser Zeit liefen in den USA und in England mehrere Entwicklungen parallel, um Rechenmaschinen zu entwickeln. Zum einen brauchte man ballistische Tabellen[4], die mittels Anfangswertlösern berechnet wurden. Dazu waren immer wieder die gleichen Rechenvorgänge nötig und es wurden viele Menschen gebraucht, diese von Hand zu berechnen. Zum Anderen wollte man den geheimen Code der Deutschen in der U-Boot-Kommunikation entschlüsseln.

Mitte der 40er Jahre entwickelte John von Neumann ein Prinzip zum Aufbau von Computern (von-Neumann-Computer), der im wesentlichen noch die heutige Rechnergeneration beschreibt (siehe Abschnitt 2.1 Aufbau des Computers). 1946 wurde der erste mit elektronischen Röhren ausgestattete Computer in Betrieb genommen (ENIAC). Die Computer dieser Generation hatten einen immensen Platzbedarf und waren extrem temperaturanfällig.

Die ersten rechnerunabhängigen Programmiersprachen wurden ab Mitte der 50er Jahre entwickelt (FORTRAN, ALGOL). Waren vorher Programmierer eher Artisten, die „ihren“ Computer beherrschten, und es schafften, die Probleme den Maschinen anzupassen, verloren sie mit den höheren Programmiersprachen etwas von ihrem Nimbus. 1955 wurde der erste Transistor-Rechner in Betrieb genommen, 1957 die ersten Magnetplattenspeicher.

In den sechziger Jahren setzten immer mehr Betriebe Computer ein. Sowohl für Banken und Versicherungen, als auch für ingenieurmäßige und wissenschaftliche Berechnungen wurden Rechner und Programme benötigt. Mit der Weiterentwicklung der Rechner wurden die Computer immer kleiner, leistungsfähiger und billiger. Damit nicht nur der einzelne Programmierer sein Programm verstehen und warten kann, wurden Standards und DIN-Normen zur strukturierten Programmierung entwickelt. Ab 1970 wurden Halbleiterspeicher entwickelt, 1971 der erste Mikroprozessor (INTEL 4004) herausgebracht.

Ende der 70er Jahre wurde das Betriebssystem UNIX entwickelt, um ein netzwerkfähiges, leistungsstarkes Betriebssystem zu haben. Rechnernetze wurden zu der Zeit insbesondere im militärischen Bereich benötigt. Man beabsichtigte mehrere Computer so zu vernetzen, dass, auch bei Ausfall einer oder mehrerer Komponenten, im Notfall noch alle wichtigen Operationen ausgeführt werden konnten. Außerdem benötigte man ein System zum schnellen Datentransfer.

Aus diesen Zeiten stammt auch das inzwischen weit bekannte „Internet“.

1977 wurden die ersten Personalcomputer (PC) herausgebracht. Anfang der Achtziger Jahre kamen jede Menge „Heimcomputer“ auf den Markt, die im wesentlichen nur zum Spielen von Computerspielen benutzt wurden. Die angebotene Software war mager, leistungsfähige Programme waren teuer. Die immer billigeren und leistungsstärkeren PCs haben in den letzten Jahren bei immer mehr Leuten Verbreitung gefunden. Sie wurden überwiegend zur Textverarbeitung genutzt und fanden so auch große Verbreitung außerhalb des mathematisch-technisch-naturwissenschaftlichen Bereiches[5].

1995 eroberte das Schlagwort „Multimedia“ die bundesdeutsche Gesellschaft. In der Zwischenzeit gibt es immer weniger Haushalte, die nicht einen Computer besitzen, sei es um die Urlaubsfotos von CD-ROM auf den Bildschirm zu laden, im Internet zu surfen, der Hausverwaltung einen Brief zu schreiben, den Anrufbeantworter und das Fax-Gerät zu ersetzen.

Im wissenschaftlichen Bereich sind vernetzte UNIX-Workstations heutzutage ein weitverbreitetes Arbeitsmittel. Ein modernes, gut organisiertes Netz kann eine Vielzahl von Softwarepaketen einer großen Zahl von Benutzern zur Verfügung stellen. Es gibt hier, gerade für wissenschaftliche Zwecke, viele leistungsfähige Programmpakete, die sogenannte „Freeware“ (kostet nichts) sind.

[4] Für jedes Geschoss musste man die Flugbahn bei verschiedenen Abschusswinkeln, Windrichtungen und -stärken kennen.
[5] Kaum ein Philosoph kann einen Artikel veröffentlichen, den er nicht auf einem gängigen Textsystem getippt hat.

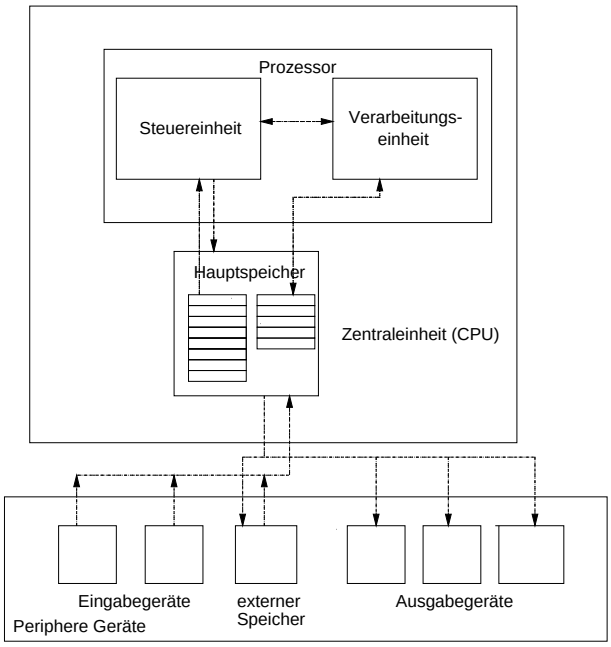


Abbildung 1: Prinzipieller Aufbau eines Computers

2 Struktur und Organisation von Computern

Das folgende Kapitel soll einen groben Über- und Einblick geben. Die verwendeten Begriffe entsprechen nicht unbedingt den Definitionen der Informatiker. Das Durchlesen ist zum grundlegenden Verständnis notwendig.

2.1 Aufbau des Computers

Computer sind Maschinen, die nur die Anweisungen ausführen, die ihnen mit Hilfe eines Programmes „beigebracht“ wurden, sie besitzen also keinerlei Intelligenz.

Im wesentlichen besteht ein Computer aus einer Zentraleinheit (Central Processing Unit: **CPU**) mit Prozessor und dem Hauptspeicher, sowie peripheren Geräten zur Ein- und Ausgabe (Input/Output oder kurz: I/O) und peripheren Speichereinheiten (z.B.: Festplatten, Diskettenlaufwerken). Diese Struktur geht noch auf den *von-Neumann-Computer* (siehe Abschnitt 1.2 Geschichte der Computer) zurück, dessen Aufbau im folgenden beschrieben wird[6]. Abbildung 1 veranschaulicht den Aufbau eines Computers.

Die **Zentraleinheit** bildet den Kern des Computers. Hier werden die Informationen verarbeitet. Ihre Komponenten sind:

- Der **Hauptspeicher**, in dem während des Programmablaufes alle für das Programm wichtigen Daten gespeichert werden, das sind sowohl Programmdaten und Verarbeitungsdaten, die andere Bestandteile bei Bedarf abrufen, als auch Zwischenergebnisse der Verarbeitung. In dem Hauptspeicher liegt auch der Speicherbereich für den Eingabe- und Ausgabepuffer. Alle Daten die ein- bzw ausgegeben werden, werden hier zwischengespeichert und weiterverarbeitet.

[6] Prinzipiell ist der Aufbau moderner Rechner gleich geblieben. Die einzelnen Komponenten werden evtl. anders benannt oder zusammengefasst.

- Der **Prozessor**. Er besteht aus:
  - der **Steuereinheit**, die den Programmablauf steuert. Dazu liest sie die Befehle sequentiell, also hintereinander, aus dem Hauptspeicher und sorgt dafür, dass diese ausgeführt werden;
  - und der **Verarbeitungseinheit**, die die Operationen nach Vorgabe der Steuereinheit durchführt und die Ergebnisse an die vorgegebene Adresse im Hauptspeicher liefert.
- Steuereinheit und Verarbeitungseinheit sind in den moderneren Computern in einem Bauteil realisiert.

über die **peripheren Geräte** findet die „Kommunikation“ des Computers mit der Umgebung statt. Man unterscheidet zwischen folgenden peripheren Geräten:

- Über die **Eingabegeräte** werden Programm- und Verarbeitungsdaten in den Hauptspeicher geladen (zum Beispiel: Tastatur, Maus).
- Auf die **Ausgabegeräte** werden die Resultate übertragen (zum Beispiel: Bildschirm, Drucker, Plotter).
- Die **externen Speicher** dienen zur Datensicherung. Sie können auch für Aus- und Eingaben verwendet werden (zum Beispiel: Festplatten, Diskettenlaufwerke, CD-ROM).

2.2 Wie arbeitet jetzt der Computer?

Ein Programm wird in den Hauptspeicher geladen und Schritt für Schritt abgearbeitet. Jede Einheit, die von dem Computer aus angesteuert werden kann, muss eine Adresse haben. Die Steuereinheit entnimmt dem Programm, von welchen Adressen Daten gelesen werden sollen, wie die Verarbeitungseinheit sie verarbeiten soll, und auf welche Adresse das Resultat geschrieben werden soll.

Jeder Speicherplatz und jedes Ein- und Ausgabegerät hat eine Adresse[7].

Man kann sich den Computer als eine Ansammlung von Schubkästen vorstellen, wobei auf jedem Schubkasten eine Adresse steht und man in diese Schubkästen Daten ablegen kann.

Ein Programm, das vom Rechner ausgeführt werden kann, muss dem Computer in „seiner“ *Maschinensprache* vorliegen. Diese Maschinensprache ist von Computer zu Computer verschieden und beinhaltet grundlegende Operationen, die der Rechner ausführen kann (z.B. das Lesen und Schreiben von Daten aus einer angegebenen Adresse, etc.). Ein Programm ist letztendlich nichts anderes, als eine Liste von aneinander hängenden Anweisungen, die nacheinander abgearbeitet werden.

2.3 Informationsdarstellung im Rechner

Im Computer können Informationen mit verschiedenen physikalischen Größen dargestellt werden, die zwei unterschiedliche Zustände annehmen können. Man wählt zur Darstellung dieser Zustände die Bezeichnung „0“ und „1“, die sogenannten **Binärzeichen**[8]. In der derzeitigen Rechnergeneration wird unterschieden zwischen:

- elektrische Ladung (Zustand „1“)/keine elektrische Ladung (Zustand „0“) in der Speicherzelle

  oder
- hohe Spannung (Zustand „1“)/niedrige Spannung (Zustand „0“) auf einer Leitung.

[7] Eigentlich „hängen“ Ein- und Ausgabegeräte an „Schnittstellen“ des Computers. Diese Schnittstellen haben wiederum Adressen.

[8] Häufig wird auch die Bezeichnung „L“(für Low) und „H“ (für High) verwendet.

Ein solches binäres Zeichen hat den Informationsgehalt von einem **Bit** (binary digit).

Um einen größeren Informationsgehalt zu erreichen, werden die Binärzeichen zu Maschinenwörtern verkettet. Die Maschinenwortlänge ist von Rechner zu Rechner verschieden. Waren vor einigen Jahren noch 8-Bit-Rechner (8 Bit = 1 **Byte**) üblich, sind in der Zwischenzeit 32-Bit- und 64-Bit-Rechner der Normalfall.

Alle Daten, die verarbeitet werden, müssen also nach bestimmten Regeln in Binärzahlen (= Dualzahlen) gewandelt werden.

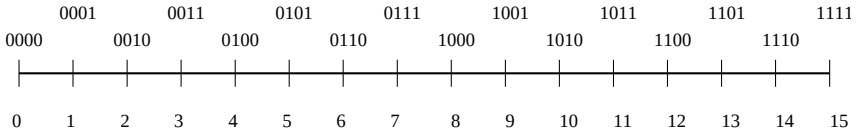
2.3.1 Zahlendarstellung

Am einfachsten lässt sich die Darstellung der ganzen Zahlen erklären. Diese Zahlen werden im Englischen als **integer** bezeichnet.

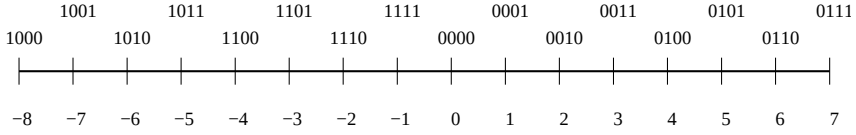
Ganze Zahlen kann man sehr einfach als **Dualzahlen**[9] darstellen. Nehmen wir an, wir haben 4 Bit zur Verfügung. Um ganze Zahlen darzustellen, sieht das folgendermaßen aus:

		dezimal		dual		
0	=	$0 \cdot 10^0$	=	0000_2	=	$0 \cdot 2^0$
1	=	$1 \cdot 10^0$	=	0001_2	=	$1 \cdot 2^0$
2	=	$2 \cdot 10^0$	=	0010_2	=	$1 \cdot 2^1 + 0 \cdot 2^0$
3	=	$3 \cdot 10^0$	=	0011_2	=	$1 \cdot 2^1 + 1 \cdot 2^0$
			...			
15	=	$1 \cdot 10^1 + 5 \cdot 10^0$	=	1111_2	=	$1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$

Wie man in der Tabelle erkennt, ist das maximal darstellbare **Intervall** mit 4 Bit $[0, 15]$, d.h. die größte darstellbare Zahl ist $2^4 - 1 = 15$. Im Zahlenstrahl kann man erkennen, dass beim Übergang von 15 nach 0 ein sog. „**Überlauf**“ stattfindet. Wenn man zu 1111_2 (15) 1 addiert, müsste eigentlich 1 0000_2 (16) herauskommen, aber da kein 5. Bit existiert, wird es abgeschnitten und es entsteht die Zahl 0000_2.



Wenn man nun das erste Bit zur Darstellung des Vorzeichens verwendet, kann man das Zahlenintevall von -2^3 bis $2^3 - 1$ ($[-8, 7]$) darstellen (s.u.). Man definiert das höherwertige Bit als Vorzeichen (1 entspricht *minus*, 0 entspricht *plus*). Die restlichen Bits werden zur Darstellung der Zahl verwendet. Da die Zahl Null nur einmal beim Übergang vom negativen in den positiven Zahlenbereich auftaucht (es gibt kein +0 oder -0), bleibt noch „Platz“ für eine weitere Zahl am Anfang des Intervalls. Die Umwandlung einer vorzeichenbehafteten 4 Bit Integer-Zahl lässt sich nun also wie folgt darstellen:



Beim Übergang vom Ende des Intervalls (7) zum Anfang des Intervalls (-8) findet das selbe statt, wie bei einer vorzeichenbehafteten Zahl: Zur höchsten Zahl 0111_2 wird 1 dazuaddiert und es entsteht 1000_2, d.h. -8. Der „Bitüberlauf“ findet in diesem Fall beim Übergang von -1 nach 0 statt.

[9] Zahlen mit der Basis 2 (also Dualzahlen) sind im Folgenden durch eine tiefgestellte 2 gekennzeichnet. Dezimalzahlen (Basis 10) sind nicht besonders gekennzeichnet.

Hinweis:

Der oben beschriebene „**Überlauf**“ führt bei der Programmierung oft zu falschen Ergebnissen, wenn man nicht auf den Wertebereich der verwendeten Variablen achtet. Will man beispielsweise mit einer vorzeichenbehafteten 32 Bit-Integer-Zahl Fakultäten berechnen, wird bereits bei $12!$ der Wertebereich **überschritten**. *Der Rechner kümmert sich darum aber nicht!* Er liefert einfach unsinnige Ergebnisse ohne Fehlermeldung. Bei der Programmierung muss also selbst darauf geachtet werden, dass der **zulässige** Wertebereich für alle Variablen eingehalten wird.

Für Interessierte die Realisierung negativer integer-Zahlen am Beispiel der Dualzahl von -1 etwas genauer:

Zunächst wird 2^n gebildet, wobei n die Anzahl der zur Verfügung stehenden bits ist. Bei 4 Bit also $2^4 = 16$.

Dann wird die Dualzahl von diesem Wert gebildet,	`1 0000`
Von dieser Zahl wird der Betrag der darzustellenden (dual-)Zahl subtrahiert:	`- 0001`
Das Ergebnis:	`1111`

Wenn man die Nullen und Einsen vertauscht und eins dazuaddiert, erhält man das gleiche Ergebnis:

positiv dezimal		pos. dual	vertauscht	negativ dual		neg. dezimal
				1000	=	-8
7	=	0111	1000	1001	=	-7
6	=	0110	1001	1010	=	-6
5	=	0101	1010	1011	=	-5
4	=	0100	1011	1100	=	-4
3	=	0011	1100	1101	=	-3
2	=	0010	1101	1110	=	-2
1	=	0001	1110	1111	=	-1
0	=	0000				

Die Null gibt es natürlich nur einmal, deshalb ist auch „Platz“ für die -8. Ein Vorteil dieser Darstellung: bei der Addition muss nicht unterschieden werden, ob es sich um positive oder negative Zahlen handelt!

Wie man gesehen hat, ist der integer-Zahlenbereich stark beschränkt. Zum einen kann man nur mit ganzen Zahlen rechnen, zum anderen ist das Intervall, in dem man rechnen kann, relativ klein.

Um den Zahlenbereich zu erweitern, gibt es die sogenannte **Gleitpunktdarstellung (floating point)**. Diese Darstellung ermöglicht einen sehr großen Wertebereich mit einer ausreichenden Genauigkeit. Grundlage der Darstellung ist, dass eine **reelle Zahl** X im Zahlensystem mit der **Basis** $B = (2, 8, 10\ oder\ 16)$ wie folgt dargestellt werden kann:

$$X = B^e m$$

Die wertbestimmenden variablen Bestandteile sind hierbei der als ganzzahlig definierte **Exponent** e und die **Mantisse** m. Die Basis B ist festgelegt. Wie die Codierung computerintern abläuft, soll hier nicht genau beschrieben werden, aber es ist offensichtlich, dass sie anders gestaltet sein muss, als bei einer `integer`-Zahl.

Beispiel:

Der Wert $X = -11.625 = -1011.101_2 = -2^3 \cdot 1.011101000..._2$ wird in einem 32-Bit-Maschinenwort folgendermaßen dargestellt:

$$\underbrace{1}_{s}\ \underbrace{10000010}_{c}\ \underbrace{01110100000000000000000}_{m^*}$$

Die Leerzeichen sind hier nur zur übersichtlicheren Gestaltung eingefügt. Das erste Bit (s) beschreibt das Vorzeichen (*engl.= sign*). c ist die Verschiebekonstante, die dafür sorgt, dass Mantisse und Exponent richtig interpretiert werden und m^* stellt die normalisierte Mantisse dar. Wer sich genauer damit auseinandersetzen möchte, dem sei hier die Lektüre entsprechender Informatik Handbüchern empfohlen.

Im Vergleich dazu noch einmal der Wert 11 (dezimal) als 32-Bit integer-Zahl:

00000000000000000000000000001011

Hieran ist klar zu erkennen, dass der Computer die ganzen Zahlen anders verwaltet, als die reellen.

2.3.2 Zeichendarstellung

Ein anderer Datentyp, der sehr oft verarbeitet werden soll, sind Schriftzeichen. Sie sind über den sogenannten **ASCII-Code**[10] verschlüsselt. Jedem Zeichen ist nach diesem Code eine Dualzahl zugewiesen.

[10] **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange. Standardisierter Code zur Darstellung von Zeichen.

3 Vom Problem zum Programm

Der Ausgangspunkt jeder Programmentwicklung ist das Problem. In diesem Kapitel soll allgemein der Weg vom Problem zum Programm dargestellt und erläutert werden. Dieser Weg vom Problem bis hin zur Lösung durch ein Programm kann in einem ersten Schritt grob in drei Phasen unterteilt werden, die nacheinander durchlaufen werden:

- Problemanalyse
- Entwurf des Algorithmus
- Erstellung des Programms

Prinzipiell ist eine weitere Gliederung der Phasen in kleineren Abschnitten möglich bzw. notwendig, um den Ablauf hinreichend genau beschreiben zu können. Ein solches Vorgehen bezeichnet man auch als schrittweise Verfeinerung, auf die in diesem Kapitel noch näher eingegangen wird.

Zunächst aber werden allgemein der Begriff Algorithmus und – wegen der besonderen Bedeutung für eine gute Programmentwicklung – die Phase der Problemanalyse erläutert.

3.1 Problemanalyse und Algorithmus

3.1.1 Zum Begriff Algorithmus

Das Wort Algorithmus ist ursprünglich abgeleitet vom Namen eines Persers (Abu Ja'far Mohammed ibn Musa **al Khowarizimi**, ca. 825 n. Chr.), der ein Lehrbuch über Mathematik geschrieben hatte. In neuen Lexika wird Algorithmus definiert als eine „spezielle Methode zur Lösung eines Problems".

Ein Algorithmus kann also – anders formuliert – verstanden werden als gemeinsamer Begriff für: Anleitung, Rezept, Vorschrift, Verhaltensmuster und Handlungsanweisung. Alle diese Begriffe stehen für eine Folge von Erklärungen „wie man etwas macht". Aus diesen Definitionen und Erklärungen wird deutlich, wie wichtig der Algorithmus für eine Problemlösung mittels Rechner ist. Denn der Rechner kann nur dann sinnvoll eingesetzt werden, wenn man ihm genaue d.h. für ihn „verständliche" Handlungsanweisungen gibt.

Ein ganz einfaches Beispiel für einen Algorithmus ist die bereits bekannte Handlungsanweisung zum Einschalten des Terminals bzw. zur LOGIN-Prozedur.

1. Terminal einschalten
2. LOGIN-Prozedur mit Angabe des USER-NAME beginnen
3. PASSWORD eingeben

Die wichtigsten Merkmale eines Algorithmus kann man schon an diesem einfachen Beispiel erkennen.

Der Algorithmus definiert einen *Prozess*. Eine wichtige Anforderung, die an den Prozess gestellt wird ist, dass er hinreichend genau ist. Das bedeutet, dass jeder der o.g. Schritte, inklusive der beabsichtigten Reihenfolge, unmissverständlich sind, so dass der *Prozessor* (in diesem Fall eine Person an einem Computerterminal) diese versteht und sie ausführen kann.

Außerdem müssen sowohl die Beschreibung des Prozesses als auch der Prozess selbst *endlich* sein.

Selbstverständlich ergibt sich als weitere Anforderung an den Algorithmus, dass der Prozess, den er beschreibt, ausführbar ist. So ist zum Beispiel die Feststellung „Die LOGIN-Prozedur benötigt in jedem Fall die Angabe eines PASSWORDS" nicht ausführbar, und stellt folglich auch keinen Algorithmus dar.

Die Ausführung des Algorithmus erfolgt schrittweise und **sequentiell** (das heißt aufeinanderfolgend). Jeder dieser Teilschritte, aus denen sich der gesamte Algorithmus zusammensetzt, kann wieder als ein

eigener (elementarer) Algorithmus angegeben werden, falls sich eine weitere Unterteilung als sinnvoll erweisen sollte.

Diese Detaillierung in weitere (Unter-) Algorithmen bedarf gründlicher Überlegung, denn es muss abgewogen werden, auf welcher Ebene (Abstraktionsebene bzw. Ebene elementarer Algorithmen) die Beschreibung des Algorithmus enden kann, ohne an Genauigkeit und Eindeutigkeit einzubüßen. Die bereits erwähnte Anweisung „Terminal einschalten“ reicht vollkommen aus. Weitere Erklärungen (Detaillierungen) sind im allgemeinen überflüssig.

Es existieren also verschiedene Abstraktionsebenen. Das Abstraktionsniveau, das heißt wie weit verfeinert wird, hängt davon ab, wie detailliert beschrieben werden muss, um noch hinreichend genau zu sein. Bei der Erstellung eines Algorithmus sollte man einige grundsätzliche Kriterien im Auge behalten:

- Voraussetzungen:
  Unter welchen Bedingungen arbeitet der Algorithmus? Was sind zulässige Ereignisse, für die der Algorithmus funktioniert? Welche Einschränkungen liegen vor und was passiert bei falscher Eingabe?
- Termination:
  Ein endlicher Algorithmus führt nicht mit Sicherheit zu einem endlichen Prozess. Ist sichergestellt, dass der Prozess für alle Eingaben endet? Ist es unmöglich, den Prozess in einen nicht endenden Zyklus zu bringen, das heißt existieren geeignete Abfragen?
- Korrektheit:
  Realisiert der Algorithmus wirklich den gewünschten Prozess, bzw. wird die gestellte Aufgabe richtig gelöst? Funktioniert der Algorithmus entsprechend der Aufgabenstellung?
- Aufwand:
  Gibt es Möglichkeiten den Algorithmus zu verbessern? Könnte ein anderer Algorithmus zur Lösung desselben Problems mit weniger Aufwand ausgeführt werden? Für einen Algorithmus, der von einem Rechner ausgeführt wird, gehört zur Aufwandsabschätzung vor allem die Betrachtung des Speicherplatz- und des Rechenzeitbedarfs.

Zusammenfassend kann gesagt werden: Ein Algorithmus beschreibt genau und endlich in meist sequentieller Form einen Prozess (die Lösung einer Aufgabe). Die Beschreibung erfolgt in bestimmten Notationen und mit Hilfe abstrakter bzw. elementarer Algorithmen. Die verwendete Notation oder algorithmische Sprache wird durch die elementaren Algorithmen (unterste Abstraktionsebene) festgelegt.

Ziel dieses Fortran95-Kurses ist die Erstellung von Programmen. Ein solches Programm ist jedoch nichts anderes als ein Algorithmus, der dadurch gekennzeichnet ist, dass der Prozessor der Rechner ist, und die Notation bzw. algorithmische Sprache die Programmiersprache Fortran ist. Die unterste Abstraktionsebene sind die Elementarbausteine von Fortran, das heißt die einzelnen Anweisungen.

Aus dem bisher über Abstraktionsebenen Gesagten ist ersichtlich, dass man einen Algorithmus zur Lösung eines Problems mittels Rechner nicht gleich in der Programmiersprache erstellen sollte. Ein komplexes Problem ist nur durch die Einführung höherer Abstraktionsebenen durchschaubar und verständlich darzustellen. Durch verfeinernde, weitergehende Formulierung von Teilalgorithmen gelangt man schließlich von einer hohen Abstraktionsebene zur niedrigen. Dies sind in der Regel dann schon Elemente der Programmiersprache.

3.1.2 Teilschritte der Problemanalyse

Da bei der Umsetzung eines Problems in ein Programm die Phase der Analyse des Problems von besonderer Bedeutung ist, wird die Problemanalyse exemplarisch in einzelne wichtige Teilschritte zerlegt. Dies soll an einem Beispiel verdeutlicht werden:

Problemanalyse:

In der ausgiebigen Diskussion über die notwendigen Maßnahmen zur Sicherung des Energiebedarfs war und ist die Nutzung der Kernenergie äußerst umstritten. Gerade in den letzten Jahren rückte dabei das Problem der anfallenden Abfallprodukte in den Vordergrund.

Problemname:

In unserem Beispiel soll auf dieses Problem eingegangen werden. Es soll versucht werden, eine Lösungsstrategie (d.h. einen Lösungsalgorithmus) – natürlich nur beispielhaft und in sehr beschränktem Rahmen – zu entwickeln. Als erster Schritt muss das Problem namentlich definiert werden. Der Problemname, der in unserem Beispiel angebracht erscheint, lautet „Atommüllbeseitigung“. Das Problem ist damit benannt und schon grob umrissen.

Problemspezifikation:

Der zweite Schritt besteht darin, die Art des Problems zu analysieren und weiter zu spezifizieren. Es gilt also festzustellen, dass die anfallenden Abfallprodukte stark radioaktiv sind und für die Umwelt eine große Gefahr darstellen. Deshalb kann der „Atommüll“ nicht, wie andere Abfallprodukte, einfach auf die Müllhalde transportiert werden.

Abgrenzung:

Die Spezifikation, dass „Atommüll“ besonders gefährlich ist, reicht natürlich nicht aus, und es muss in einem nächsten Schritt versucht werden, das Problem weiter abzugrenzen. Die Abgrenzung besteht darin, dass unter „Atommüll“ in unserem Beispiel nur die abgenutzten Brennstäbe verstanden werden sollen. Unberücksichtigt bleiben andere zusätzlich entstehende Abfallprodukte (zum Beispiel radioaktives Kühlwasser). Des Weiteren soll die Menge des hochgiftigen Abfalls begrenzt sein.

Ziel:

Für eine Lösungsstrategie sind die bisherigen Schritte bei weitem noch nicht ausreichend, da überhaupt kein Lösungsziel angegeben wurde. Bezüglich der Atommüllbeseitigung existieren seitens der „Verantwortlichen“ eine Vielzahl von Zielen, welche die Strategie festlegen, als da wären: Wiederaufarbeitung und Entsorgung, Endlagerung in unterirdischen Mülldeponien usw. Eine weitere Möglichkeit, die bereits diskutiert wurde, ist die Errichtung von Endlagerstätten auf anderen Planeten (zum Beispiel auf dem Mond). Dieses Ziel soll in unserem Beispiel weiterverfolgt werden.

Mittel:

Die Ziele von Lösungsstrategien hängen natürlich in entscheidendem Maße von den Mitteln ab, die zur Verfügung stehen. Dazu gehören Kapital, Personal, Material, Maschinen, technisches Know How, usw. Da die Mittel im Allgemeinen begrenzt sind, muss diesem Teilschritt besondere Bedeutung beigemessen werden.

Entscheidung:

Die vorangegangenen Teilschritte Abgrenzung, Ziel und Mittel haben das Problem soweit transparent gemacht, dass eine Entscheidung für einen bestimmten Lösungsweg möglich wird. Gehen wir in unserem Beispiel mal davon aus, dass genügend Kapital, Material und technisches Know How zur Verfügung stehen. Unter diesen sehr günstigen Umständen kann eine Endlagerung der giftigen Abfallstoffe auf einem fremden Planeten ernsthaft in Betracht gezogen werden. Die Entscheidung versetzt uns nun in die Lage, eine grobe Handlungsanweisung bzw. einen Lösungsalgorithmus zu entwerfen.

Dieser erste grobe Algorithmus wird dann in weitere Teilschritte unterteilt, d.h. schrittweise verfeinert. Ein entscheidender Teilalgorithmus, der entwickelt werden muss, betrifft die Berechnungen der Rake-

tenflugbahn von der Erde zum anderen Planeten. Diese Berechnungen werden sicher mit Hilfe eines Rechnerprogramms durchgeführt. Das verwendete Programm stellt dabei einen weiteren Unteralgorithmus dar.

Zum Schluss sei die Untergliederung der Problemanalyse in die einzelnen Teilschritte noch einmal graphisch mittels eines Struktogramms (dazu später mehr) in Bild 2 dargestellt.

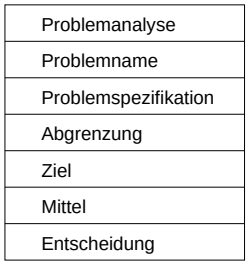


Abbildung 2: Struktogramm Problemanalyse

Grundsätzlich ist natürlich eine Problemanalyse denkbar, die eine ganz andere Handlungsanweisung (Algorithmus) ergibt. Der Algorithmus zur Endlagerung auf anderen Planeten ist überflüssig, wenn man bei der Analyse des Problems zu dem Schluss kommt, dass das Problem „Atommüllbeseitigung“ nur durch den Verzicht auf Kernenergie gelöst werden kann. Hieraus wird deutlich, welche entscheidende Bedeutung der Problemanalyse zukommt. Auch bei den kleinen numerischen Problemen innerhalb dieses Kurses sollte deshalb eine Problemanalyse durchgeführt werden.

3.2 Prinzipieller Ablauf bei der Erstellung eines Programms

Wie bei jeder Aufgabe, steht auch am Anfang jeder Programmentwicklung immer ein Problem. Im Rahmen dieses Fortran-Kurses werden ausschließlich numerische Aufgaben behandelt; folglich haben wir es auch nur mit numerischen Problemen zu tun. Der Weg vom Problem bis zur Lösung durch ein fertiges Programm ist ziemlich lang, und sollte deshalb in mehrere Phasen unterteilt werden.

Die einzelnen Phasen werden im folgenden dargestellt, um die Vorgehensweise zu veranschaulichen. Dabei werden zum Teil bereits Begriffe und Dokumentationstechniken verwendet, die erst später erläutert werden. Man lasse sich davon aber nicht irritieren, weil sie das grundsätzliche Verständnis nicht behindern.

Problemanalyse:

Das Problem (die Aufgabenstellung) wird zunächst benannt, untersucht und abgegrenzt, mit dem Ziel, die Eingangsdaten (Voraussetzungen) und die notwendigen bzw. erwünschten Ergebnisse möglichst genau zu beschreiben. Allgemein gehört dazu die Beschreibung des Ziels (was soll berechnet werden?) und die Sichtung der Mittel, die zur Verfügung stehen (zum Beispiel Näherungsverfahren, Computer usw.). Mit ihnen und der Abgrenzung (vgl. Seite 11) müssen die Entscheidungen getroffen werden, die zur Problemlösung führen.

Entwurf des Algorithmus:

Die Problemanalyse schafft die Grundlagen zur Entscheidung für das Lösungsverfahren. Beim Entwurf der Algorithmen wird dieses Verfahren näher beschrieben und dabei, gegebenenfalls mittels Verfeinerungen, weiterentwickelt. Dabei bedient man sich eines graphischen Hilfsmittels und zwar der Struktogramme (mehr dazu später). Die Entwicklung des Algorithmus bzw. seine Verfeinerungen sollten so weit gehen, dass ein Entwurf entsteht, der sich möglichst direkt in die Programmiersprache übertragen lässt.

Erstellung des Programms (Programmierung):

Der Algorithmus wird aus der sprachlichen und/oder graphischen Darstellung (Notation) in die Programmiersprache übertragen. Die Darstellung des Algorithmus ist im Allgemeinen so abstrakt, dass die Programmiersprache beliebig gewählt werden kann. In diesem Kurs steht am Ende der Programmierung immer ein Fortran-Programm.

Da die Programmierung bzw. das Erlernen desselben Hauptziel dieses Kurses ist, wird die zuletzt erwähnte Phase „Erstellung des Programms“ in weitere Teilschritte zerlegt, um den Arbeitsablauf zu verdeutlichen:

Editieren:

Das Programm wird zunächst vom Papier („handschriftliche Form“) mit Hilfe des bereits bekannten Editors auf den Bildschirm bzw. das Terminal übertragen (es steht damit gleichzeitig in einem internen Speicher des Rechners). Das so erstellte (editierte) Programm kann auf eine Datei geschrieben und gespeichert werden.

Übersetzen (Compilieren) :

Für den Rechner ist es notwendig, dass das editierte Programm von einem **Compiler** (Übersetzungsprogramm) in die für ihn „verständliche“ Maschinensprache (Binärcode) übersetzt wird. Dies geschieht durch den Aufruf des Fortran-Compilers, der zunächst eine sog. Objektdatei erstellt (**.** o-Datei). In einem zweiten Schritt wird dann die Objektdatei **gelinkt**, d.h. es werden weitere Objektdateien bzw. Bibliotheken dem eigentlichen Programm hinzugefügt, die dieses für seine spätere Ausführung benötigt. Der Linker wird i.A. vom Compiler automatisch gestartet, so dass dies nicht vom Hand geschehen muss. Das Ergebnis des Linkens ist ein vom Betriebssystem ausführbares Programm.

Fehlerkorrektur:

Der Compiler erkennt beim Übersetzen des Programms formale Fehler, die gegen die **Syntax** der Programmiersprache verstoßen, und listet sie in einer speziellen Liste, dem sogenannten Compiler-Listing, auch auf. So ist es möglich, die syntaktischen Fehler des Programms auf komfortable Art und Weise zu erkennen. Mit Hilfe des Editors können die Fehler korrigiert werden. Von dort wird dann wieder neu übersetzt.

Laden und Ausführen des Programms:

Treten bei der Übersetzung keine formalen Fehler mehr auf, so kann das Programm durch Eingabe seines Namens in der Kommandozeile gestartet werden. Es wird dann entsprechend seinem Aufbau abgearbeitet. Dabei können wiederum Fehler auftreten, die sogenannten **Laufzeitfehler**. Die Korrektur macht wieder einen Rückgriff auf die Editierphase notwendig, denn das Programm muss ja in der Quellenform geändert bzw. korrigiert werden.

Programmtest:

Nach Beseitigung der beiden Fehlertypen Syntax- und Laufzeitfehler läuft das Programm im Rechner störungsfrei. Es ist aber mit Testdaten noch nachzuweisen, ob das Programm auch tatsächlich das macht, was man von ihm erwartet. Bei mathematischen Problemen ist die richtige Berechnung von Testwerten aufzuzeigen. Hierdurch werden im Zweifelsfalle die „**logischen**“ Fehler des Programms aufgedeckt.

Dokumentation des Programms:

Die Dokumentation eines Programms sollte so verfasst sein, das auch sogenannte „Unbedarfte“, die Programmstruktur ohne große Schwierigkeiten erkennen und das Programm in seinem Ablauf

nachvollziehen können. Dies ist im Wesentlichen durch eine gute Kommentierung im Programm selbst und durch korrekte Struktogramme zu erreichen.

3.3 Dokumentationstechniken (Struktogramme)

Die Darstellung des Algorithmus erfordert eine Notation, auf die man sich einigen und dann festlegen muss, um auch anderen den Algorithmus transparent zu machen. Hierbei gibt es natürlich mehrere Möglichkeiten, mit deren Hilfe Algorithmen klar (anschaulich) und korrekt dargestellt werden können. Unter „korrekt“ wird an dieser Stelle verstanden, dass der Algorithmus genau das leistet, was beabsichtigt ist.

Ein Lernziel dieses Kurses ist die grafische Darstellung der Algorithmen mit Hilfe der Struktogramme, die nach ihren Erfindern auch **Nassi-Shneiderman-Diagramme** genannt werden. Diese Art der Darstellung soll in diesem Kapitel ausführlich erläutert werden. Vor Beginn der Programmierung soll in diesem Kurs auch für jedes Programm ein Struktogramm angefertigt werden. Alle Algorithmen haben die gleichen elementaren Grundstrukturen, die zu ihrer Darstellung benötigt werden. Diese Grundstrukturen sind:

1. Sequenz: Die Aneinanderreihung von Teilalgorithmen
2. Alternation: Auswahl unter möglichen Teilalgorithmen
3. Iteration: Wiederholung von Teilalgorithmen

Diese Grundstrukturen reichen aus, um alle sequentiellen Algorithmen zu entwerfen. Bei der Programmierung werden überwiegend sequentielle Abläufe verwendet. In diesem Kurs sollen deshalb ausschließlich sequentielle Algorithmen verwendet werden.

Im weiteren Verlauf des Skriptes werden die Algorithmen nach diesen elementaren Grundstrukturen entwickelt. Das Ziel dabei ist, dass auch die mit Hilfe dieser Algorithmen entwickelten Programme nur noch diese Strukturen aufweisen. Programme, die auf diese Weise erstellt werden, sind in ihrem Aufbau und ihrer Logik viel leichter überschaubar. Das trägt sehr zur Transparenz der Programme bei. Diese Art der Programmentwicklung wird auch „strukturiertes Programmieren“ genannt. Im folgenden werden die Grundstrukturen der Algorithmen näher erläutert und ihre Darstellung im Nassi-Shneiderman-Diagramm erklärt.

Die Struktogramme stellen die Teilalgorithmen oder Segmente in Form von Rechtecken dar. Bei der Sequenz mehrerer Segmente werden die entsprechenden Rechtecke übereinandergesetzt. Dabei ist der Ausgang des Einen der Eingang des Nächsten. Pfeile und Ähnliches entfallen, da die Abfolge der Segmente stets von oben nach unten geht.

3.3.1 Sequenz

Unter Sequenz versteht man eine Aneinanderreihung von Teilalgorithmen oder Segmenten, die als unteilbares Grundkonstrukt aufgefasst werden können. Diese Segmente sind Programmfragmente (entweder einzelne Anweisungen oder auch eine Folge mehrerer Anweisungen). Solche Segmente zeichnen sich dadurch aus, dass sie einen genau definierten Eingang und einen genau definierten Ausgang besitzen. In Bild 3 ist ein Algorithmus zu sehen, der sich aus n Teil- oder Unteralgorithmen „Anweisung 1“, „Anweisung 2“,..., „Anweisung n“ zusammensetzt.

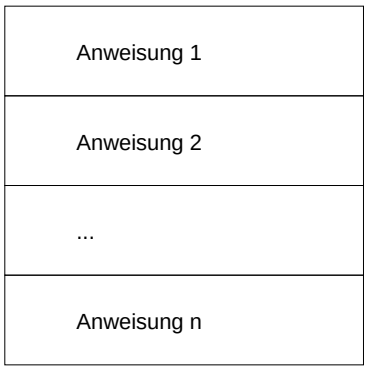


Abbildung 3: Struktogramme – Sequenz

Bei dieser Darstellung sind die Proportionen der Rechtecke ohne Bedeutung; sie werden dem Platzbedarf angepasst, der für den einzufügenden Text benötigt wird. Lediglich aneinanderstoßende Ober- bzw. Unterkanten der Rechtecke müssen gleich lang sein.

3.3.2 Alternation – Alternative Auswahl

Die Alternation stellt eine Auswahl von Teilalgorithmen, unter der Abfrage einer Bedingung, dar. Die Grundform ist dabei die alternative Auswahl. Sie wählt in Abhängigkeit einer Bedingung, die wahr oder falsch sein kann, den entsprechend vorgesehenen Teilalgorithmus aus. Ist eine Bedingung erfüllt, so ist sie „wahr“ und nur der Teilalgorithmus „Anweisung 1“ soll ausgeführt werden. Im anderen Fall ist die Bedingung nicht erfüllt, also „falsch“, und der Teilalgorithmus „Anweisung 2“ soll ausgeführt werden. Die alternative Auswahl kann nun im Struktogramm, wie in Bild 4 zu sehen, dargestellt werden.

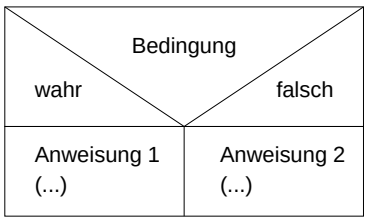


Abbildung 4: Struktogramme – Alternative Auswahl

3.3.3 Alternation – Bedingte Auswahl

Eine weitere – spezielle – Form der Alternation ist die bedingte Auswahl. Diese entscheidet in Abhängigkeit von einer Bedingung, ob ein Teilalgorithmus bzw. weitere Programmsegmente ausgeführt werden sollen oder nicht. Ist die Bedingung erfüllt („wahr“) wird ein Teilalgorithmus ausgeführt. Formal ist die bedingte Auswahl ebenfalls eine Alternation, bei der lediglich bei nicht erfüllter („falsch“) Bedingung kein Teilalgorithmus ausgeführt wird. Das entsprechende Nassi-Shneiderman-Diagramm stellt sich für die bedingte Auswahl wie in Bild 5 dar.

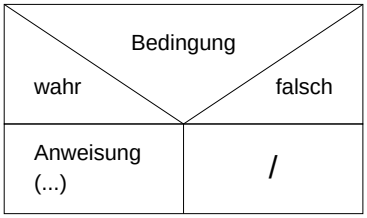


Abbildung 5: Struktogramme – Bedingte Auswahl

Sowohl die alternative als auch die bedingte Auswahl spielen bei der Entwicklung numerischer Lösungsalgorithmen eine große Rolle. Diese „**Kontrollstrukturen**“ bilden nahezu den Kern eines jeden Programms. Eine bedingte Auswahl bietet sich besonders an, um sogenannte „verbotene“ Operationen (zum Beispiel Division durch Null) auszuschließen oder gegebenenfalls einen vorgegebenen Definitonsbereich einzuhalten.

Beispiel: Sollen in einem Programm ausschließlich reelle Quadratwurzeln berechnet werden, kann mit Hilfe der bedingten Auswahl das Berechnen der Wurzel nur dann durchgeführt werden, wenn die Zahl, deren Wurzel bestimmt werden soll, nicht negativ ist.

3.3.4 Alternation – Mehrfachauswahl

Die Mehrfachauswahl ist eine Erweiterung der einfachen Alternative. Es werden mehrere Alternativen berücksichtigt. Jede der möglichen auswählbaren Aktionen wird durch einen Kennwert charakterisiert. Trifft keiner der Kennwerte zu, wird eine sonstige Handlung ausgeführt. Im Struktogramm wird die Mehrfachauswahl wie in Bild 6 zu sehen dargestellt.

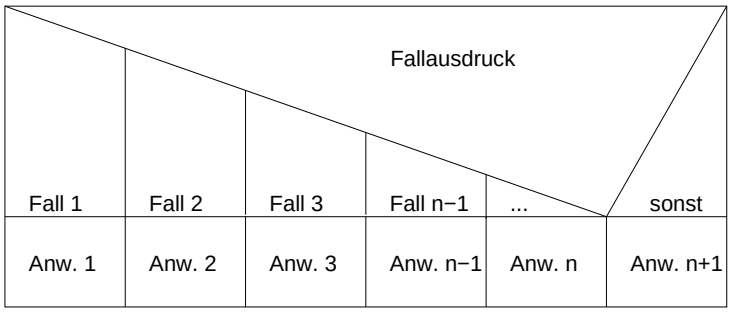


Abbildung 6: Struktogramme – Mehrfachauswahl

3.3.5 Iteration (Wiederholung)

Die Iteration ist die Wiederholung eines Teilalgorithmus in Abhängigkeit von einer Bedingung. Dabei sind verschiedene Formen möglich; eine Wiederholung mit Eintrittsbedingung (kopfgesteuert), eine Wiederholung mit Austrittsbedingung (fußgesteuert) und eine Wiederholung mit vorgegebener Zählvorschrift (Zählschleife).

3.3.6 Wiederholung kopfgesteuert

Am Anfang einer solchen Iteration steht eine Eintrittsbedingung, die überprüft wird. Ist die Bedingung erfüllt (wahr), wird der anschließende Teilalgorithmus so lange wiederholt, bis sie nicht mehr erfüllt (falsch) ist. Für den Fall, dass die Bedingung von Beginn an nicht erfüllt ist, gelangt das vorgesehene Programmsegment nicht zur Ausführung. Es findet also eine richtige Eingangskontrolle statt. Das Struktogramm hierfür ist in Bild 7 zu sehen.

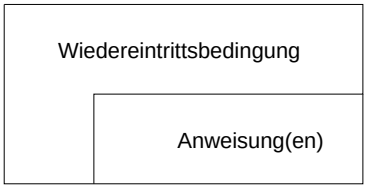


Abbildung 7: Struktogramme – Wiederholung kopfgesteuert

3.3.7 Wiederholung fußgesteuert

Bei dieser Form der Wiederholung steht der Teilalgorithmus (Anweisung) vor der (Austritts-) Bedingung. In diesem Fall wird, im Gegensatz zur Iteration mit Eintrittsbedingung, der Teilalgorithmus so lange wiederholt, bis die Bedingung zum „Austritt“ erfüllt ist. Zu beachten ist bei dieser Art der Wiederholung, dass der Teilalgorithmus mindestens einmal ausgeführt wird. Das Struktogramm hierfür ist in Bild 8 zu sehen.

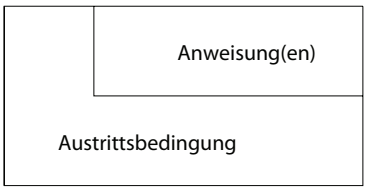


Abbildung 8: Struktogramme – Wiederholung fußgesteuert

3.3.8 Wiederholung mit Zählvorschrift (Zählschleife)

Bei der Wiederholung mit Zählvorschrift wird der Teilalgorithmus nicht aufgrund eines logischen Ausdrucks (Bedingung) wiederholt, sondern durch das Verändern einer Zählvariablen gesteuert. Einer solchen Zählvariablen wird dabei zunächst ein Anfangswert zugewiesen, der ständig pro Wiederholung um eine ebenfalls vorgegebene Schrittweite erhöht wird, bis der festgesetzte Endwert erreicht ist. Sind Zählvariable und Endwert gleich groß, wird noch einmal wiederholt. Erst wenn die Zählvariable größer als der Endwert ist, erfolgt keine Ausführung des Teilalgorithmus. Das Nassi-Shneiderman-Diagramm ist in Bild 9 dargestellt.

Abbildung 9: Struktogramme – Zählschleife

Die drei Grundstrukturen Sequenz, Alternation und Iteration reichen – wie bereits zu Beginn des Kapitels erwähnt – im Prinzip aus, um für alle Probleme einen Lösungsalgorithmus zu formulieren.

Bei größeren Problemen, die komplexere Lösungsalgorithmen verlangen, muss dieser natürlich Schritt für Schritt in kleinere Teilalgorithmen unterteilt und verfeinert werden, bis letztlich hin zu den Grundstrukturen. Diese können dann mit relativ wenig Aufwand in die Programmiersprache Fortran übertragen werden.

Hier sei angemerkt, dass gute, schrittweise verfeinerte Algorithmen die Programmierung selbst sehr erleichtern. Außerdem hat eine solche Vorgehensweise den Vorteil, dass die Programme zwangsläufig gut aufgebaut sind, da ihr Aufbau gründlich durchdacht ist.

4 Grundelemente der Programmiersprache Fortran95

4.1 Vorbemerkungen zur verwendeten Schreibweise

Es sollen folgende Layout-Konventionen gelten:

- Normaler Text.
- `Fortan95 Programmbeispiele erscheinen in Schreibmaschinenschrift.`
- *Begriffe im Text, die hervorgehoben werden sollen, erscheinen kursiv.*
  Mathematische Ausdrücke ebenfalls.
  In formalen Fortran95-Regeln zeigt die kursive Schreibweise an, dass hier irgendetwas vom Benutzer problemangepasst ausgewählt werden soll (Datentypen, Variablennamen, ...).
- Angaben in eckigen Klammern [ ] innerhalb einer formalen Fortran95 Regel sind optional und können weggelassen werden.
- Drei Punkte ... innerhalb einer formalen Fortran95-Regel bedeuten, dass das davorstehende Sprachelement in eckigen Klammern beliebig oft wiederholt werden kann.

Zwei Beispiele:

Eine `real`-Anweisung zur Definition zweier Real-Variablen x und $wert$:

```
real :: x, wert
```

Die formale Regel der `real`-Anweisung zur Definition von Konstanten. Das Attribut *Parameter* gibt an, dass es sich um eine Konstante handelt, die im Programm nicht mehr geändert werden kann:

```
real, parameter :: v=zahl [, v=zahl ] ...
```

4.2 Das Henriette Programm

Wir stellen nun ein vollständiges Fortran95-Programm vor, es heißt Henriette. Es berechnet die Fläche eines Kreises, nachdem es uns vorher zur Eingabe des Radius aufgefordert hat. Henriette soll daneben auch die wichtigsten Sprachelemente und den allgemeinen Aufbau eines Fortran95-Programms demonstrieren.

Ein Fortran95-Programm besteht aus Anweisungen, die der Rechner nacheinander ausführen soll. Damit später bei den Erklärungen einzelne Anweisungen zitiert werden können, sind im Beispielprogramm die Zeilen durchnumeriert.

Hier ist nun das „Listing“ des Henriette-Programms:

```
 1. ! Das ist das Program Henriette
 2. ! Es soll die Grundelemente eines Fortran95-Programms zeigen
 3. !
 4. ! Programmanweisung, keine implizite Typenvereinbarung
 5. program henriette
 6. implicit none
 7.
 8. ! Deklarationsteil
 9.
10. integer :: n
11. real :: kreisflaeche, radius, wert
12. real, parameter :: pi=3.1415
13.
14. ! Ende des Deklarationsteils
15.
16. ! Einlesen des Radius mit Ausgabe auf dem Bildschirm
17.
18. print *, 'Jetzt den Radius eingeben !'
19. print *
20. read *, radius
21.
22. ! Ende des Einlesens
23.
24. ! jetzt folgen die Anweisungen und Berechnungen
25.
26. n = 2
27. kreisflaeche = pi * radius**n
28. wert = 2 * pi * radius
29.
30. ! Ausgabe der Ergebnisse
31.
32. print *, 'Radius = ',radius
33. print *, 'Kreisflaeche = ',kreisflaeche
34. print *, 'Umfang = ',wert
35.
36. ! Programmende
37.
38. end program henriette
```

4.3 Fortran95-Elemente im Henriette-Programm

Zeichensatz:

Zur Notation einer Sprache benötigt man Zeichen. In Tabelle 1 sind die zulässigen Zeichen für die Fortran95-Sprache dargestellt. Alle Befehle des Beispielprogramms sind aus diesen Zeichen aufgebaut. Zwischen Groß- und Kleinbuchstaben wird nicht unterschieden.

Die 26 Buchstaben	a b c d e f g h i j k l m n o p q r s t u v w x y z
Die 10 Zahlen	0 1 2 3 4 5 6 7 8 9
21 Sonderzeichen	= + - * / ( ) , . > < ; : $! ? & _ ' "
Das Leerzeichen:	␣

Tabelle 1: Zeichensatz für Fortran95-Programme

Kommentare:

Kommentare haben keinen Einfluss auf das Programm und werden beim Übersetzen (*compilieren*) in ein ausführbares Programm einfach ignoriert. Sie sind jedoch unerlässlich zur Dokumentation und für die Übersichtlichkeit des Programmes. Kommentare werden durch ein Ausrufungszeichen „!“ eingeleitet. Mit dem gerade Gesagten ist klar, dass die ersten vier Zeilen des „Henriette“-Programms nur Kommentarzeilen sind. Auch Leerzeilen können ein Programm übersichtlicher machen. Sie haben ebenfalls keine Auswirkung auf den Programmablauf (Zeile 3).

`program`-Anweisung, Programmname:

Die 5. Zeile ist die erste Zeile mit Text, aber ohne ein „!“ in der ersten Spalte. Es ist die `program`-Anweisung, die dem Programm einen Namen gibt und gleichzeitig den Beginn des Programms kennzeichnet. Allgemein hat diese Anweisung die Form:

```
program name
```

Diese Zeile steht immer am Beginn eines Programms, davor dürfen sich nur Kommentarzeilen befinden. *name* ist ein von der Programmiererin gewählter Ausdruck. Er darf max. 31 Zeichen lang sein und Buchstaben oder Zahlen aus dem Fortran95-Zeichensatz enthalten. Außer dem Unterstrich darf kein Sonderzeichen verwendet werden[11]. Der Name muss mit einem Buchstaben beginnen.

Anmerkung: Nicht nur Programme, sondern auch Variablen, Felder und Unterprogramme werden mit Namen bezeichnet. In den folgenden Kapiteln werden diese Begriffe genauer erklärt, hier geht es nur darum, dass alle diese Namen nach der eben beschriebenen Regel gebildet werden.

Übung:

Sind folgende Fortran95-Namen zulässig?

```
x
h12v3
y3.2
her_bert
günaydın
```

Variablen, Datentypen:

Die Zeilen 10, 11 und 12 sind die `real`- und `integer`-Variablendeklarationen. Bevor diese Anweisungen erklärt werden noch einige Anmerkungen:

Es wurde bereits gesagt, dass sich die Programmiersprache Fortran95 stark an die mathematische Formelsprache anlehnt. Das Beispielprogramm ist wahrscheinlich auch für jemanden, der noch nie etwas von Fortran95 gehört hat, weitgehend „lesbar“ , d.h. er kann mit Hilfe der eingestreuten Kommentarzeilen den Rechenweg nachvollziehen. Das sollte aber nicht dazu führen, dem Rechner mathematisches

[11] auch kein Leerzeichen.

Verständnis zuzutrauen, nur weil er mathematische Formeln auswerten kann. Der Rechner kann Inhalte von verschiedenen Speichern transportieren und mit denen anderer verknüpfen, Rechnen im Sinne von Ableiten oder Gleichungen auflösen kann er nicht, er kann auch nicht den Sinn von Formelausdrücken überprüfen.

In der Fortran95-Sprache macht sich dieser Unterschied unter anderem dadurch bemerkbar, dass man angegeben muss, ob eine auftretende numerische Größe als **reelle** oder **ganze Zahl** interpretiert werden soll. In der Umgangssprache wird hier keine strenge Unterscheidung gemacht. Dazu ein Beispiel: Der Durchmesser einer Maschinenwelle ist mit 10 mm angegeben. Er könnte natürlich auch 9,9 oder 10,5 mm betragen. Wir wissen, dass er einen Wert aus dem Bereich der reellen Zahlen annehmen kann.

Geben wir hingegen an, dass sich in einer Kiste 10 Maschinenwellen befinden, so hat die Zahl 10 eine andere Bedeutung: Für die Anzahl von Wellen kommen nur Werte aus dem Bereich der ganzen Zahlen in Frage, denn 9,9 Maschinenwellen gibt es nicht.

Der Rechner muss aber wissen, welche Werte eine verwendete Variable annehmen kann. Grund dafür ist die unterschiedliche interne Darstellungsweise von reellen und ganzen Zahlen. Ganze Zahlen können im Zahlensystem auf der Basis 2 dargestellt werden, reelle Zahlen werden durch Mantisse und Exponent ausgedrückt. (Vgl. Abschnitt 2.3.1 Zahlendarstellung)

Nun können Zeile 10 und 11 erklärt werden. Durch die formale Anweisung:

```
integer :: v [,v ] ...
```

gibt man dem Rechner an, dass die Variable(n) *v* Werte aus dem Bereich der ganzen Zahlen annehmen kann (können). *Integer* ist das englische Wort für *ganz*.

Fortran95 kennt zwei reelle Datentypen: `real` und `double precision`. Mit der formalen Anweisung:

```
real :: v [,v ] ...
```

gibt man dem Rechner an, dass die Variable(n) *v* Werte aus dem Bereich der reellen Zahlen annehmen kann (können). Die Ziffern werden dabei mit einem Dezimalpunkt getrennt, nicht mit einem Dezimalkomma.

Für die Darstellung einer reellen Zahl mit dem Typ `double precision` wird die doppelte Anzahl Bytes reserviert. Die Zahl kann mit erhöhter, annähernd doppelter Genauigkeit dargestellt werden. Die Deklaration von Variable(n) mit einem größeren Genauigkeitsbereich erfolgt ganz analog mit der formalen Anweisung:

```
double precision :: v [,v ] ...
```

v ist ein selbstgewählter Name, der nach den schon beschriebenen Regeln gebildet wird. Gute Programme erkennt man unter anderem auch daran, dass die gewählten Namen einen Sinn ergeben. Also wie im Beispielprogramm die Vereinbarung von „`radius`“ anstelle von „`r`“, oder von „`kreisflaeche`“ anstelle von „`wert`“.[12]

Durch die `real`- und die `integer`-Anweisung wurde festgelegt, dass die Variablen `kreisflaeche`, `radius`, `wert` und `pi` Werte aus dem Bereich der reellen Zahlen, die Variable `n` dagegen nur Werte aus dem Bereich der ganzen Zahlen annehmen kann.

[12] Zusätzlich, oder zumindest dann, wenn die Variablennamen nicht selbsterklärend gewählt werden können, kann ein Kommentar sinnvoll sein.

Implizite Typvereinbarung:

Im vorherigen Abschnitt ging es um die Bedeutung selbsterklärender Variablennamen. Fortran95 ermöglicht es leider, in einigen Fällen ganz auf eine Variablendeklaration zu verzichten. Diese Möglichkeit kann man aber – und in allen Programmen soll das auch so gemacht werden – explizit ausschließen.

Wenn Variablen im Programm verwendet werden, die nicht durch eine `integer`-, `real`- oder `double precision`-Anweisung explizit einem der oben erwähnten Datentypen zugeordnet wurden, dann gilt die implizite Typkonvention: Alle Variablen, deren Namen mit einem i, j, k, l, m oder n beginnen, werden als Integer-Variablen interpretiert. Variablen, deren Namen mit irgendeinem anderen Buchstaben des Alphabets beginnen, werden als Real-Variablen interpretiert.

Diese Regelung soll nicht angewendet werden. Genauer: Es soll jede im Programm verwendete Variable explizit deklarieren und die implizite Typkonvention ausgeschlossen werden.

Dazu dient die `implicit none`-Anweisung (Zeile 6). Mit dieser Anweisung wird vereinbart, dass der Typ der Datengrößen nicht vom Anfangsbuchstaben abhängt:

```
implicit none
```

Diese Anweisung muss vor dem Deklarationsteil stehen.

Zuweisung:

In der Fortran95-Sprache gibt es verschiedene Möglichkeiten, den Zahlenwert einer Variablen zu definieren. Um den einfachsten Fall einer Wertzuweisung darzustellen, überspringen wir jetzt einige Zeilen im Beispielprogramm und gehen zu Zeile 26:

```
n = 2
```

Diese Anweisung erscheint möglicherweise so selbstverständlich, dass man keine Worte darüber zu verlieren braucht. Der Rechner versteht diesen Befehl folgendermaßen:

Durch das „=“-Zeichen wird der Variablen auf der linken Seite des Gleichheitszeichens, in diesem Fall n der Wert auf der rechten Seite des Gleichheitszeichens, in diesem Fall die Integer-Konstante 2, zugewiesen. Nach der Ausführung hat n den Wert 2 angenommen.

Das „=“-Zeichen hat in Fortran95 eine andere Bedeutung als in der Mathematik. Mathematisch ist die Aussage

$$n = n + 1$$

natürlich immer falsch. In Fortran95 dagegen ist die Anweisung

```
n = n + 1
```

durchaus sinnvoll. Nach Ausführung dieser Anweisung hat sich der Wert der Variablen n um den Wert der Integer-Konstanten 1 erhöht.

Konstanten, `parameter`-Attribut:

Zur Erklärung des Begriffs Konstante: Variablen können im Laufe des Programmablaufs verschiedene Werte annehmen, Konstanten haben immer den gleichen Wert. Integer-Konstanten sind ganze Zahlen ohne Dezimalpunkt. Real-Konstanten sind Zahlen mit Dezimalpunkt.

Durch verschiedene „=“-Anweisungen kann eine Variable während des Programmablaufs verschiedene Werte annehmen. Manchmal braucht man im Programm immer den gleichen Zahlenwert, wie im Beispiel den Wert von π (=3.1415). Da dieser Wert ganz sicher während der Programmausführung nicht verändert werden soll, können wir die Real-Variable `pi` als Konstante deklarieren. Die dazugehörige formale Anweisung lautet:

```
real, parameter :: v=e [,v=e ] ...
```

wobei *v* für einen selbstgewählten Namen und *e* für einen konstanten Wert steht.

Der doppelte Doppelpunkt ist in allen bisher vorgestellten Variablendeklarationen enthalten und trennt die eigentlichen Datengrößen – die Konstanten – von dem vereinbarten Datentyp – `real` – und eventuell angegebenen *Attributen*. Das hier verwendete Attribut `parameter` kennzeichnet, das es sich um Konstanten handelt.

Das folgende Beispiel zeigt noch einmal die Anwendung dieser Anweisung:

```
! Typ der Konstanten pi und e wird als double precision deklariert
! und die Zahlenwerte von pi und e werden festgelegt

double precision, parameter :: pi=3.1415, e=2.71828
.
.
.
! Es wird versucht den Wert von e zu verändern !

e = 3.5

! An dieser Stelle bricht der Compiler mit einer Fehlermeldung ab !
```

Das `parameter`-Attribut macht aus Variablen Konstanten. Das schützt vor einer versehentlichen Veränderung dieser Größe und erhöht die Lesbarkeit von Programmen.

Ein- und Ausgabe:

Zeile l8 und 20 enthalten zwei neue Elemente der Fortran95-Sprache, und zwar die `print *`- und die `read *`-Anweisung. Durch die `print`-Anweisung wird der Rechner angewiesen, Informationen oder Rechenergebnisse zunächst auf den Bildschirm (das Terminal) auszugeben. Durch die `read`-Anweisung werden dem Rechner Daten, die wir mit Hilfe der Tastatur zunächst ebenfalls auf den Bildschirm schreiben, übermittelt. Durch die beiden Anweisungen besteht die Möglichkeit, während des Programmablaufs mit dem Rechner zu kommunizieren.

Die allgemeine Form der `print`-Anweisung lautet:

```
print * [,v ] ...
```

wobei *v* sein kann:

- eine `integer`-, eine `real`- oder eine `double precision`-Konstante (also eine Zahl)
- der Name einer Variablen
- eine Textkonstante
- ein Arithmetischer Ausdruck (siehe nächster Abschnitt)

Unter einer Textkonstanten versteht man eine in Anführungsstriche eingeschlossene Zeichenfolge.

Zur weiteren Erklärung des `print`-Befehls einfache Beispiele:

```
print *, ' pi = ', 3.1415
```

erzeugt die Ausgabe:

```
pi =  3.141499996
```

Die Anweisungen:

```
x = 10.5
print *, 'Variable x = ', x, ' ! '
```

erzeugen die Ausgabe:

```
Variable x =  10.50000000 !
```

Bei jedem `print`-Befehl wird grundsätzlich am Beginn einer neuen Zeile des Terminals mit der Ausgabe begonnen. Die Anweisung:

```
print *
```

erzeugt eine Leerzeile auf dem Bildschirm.

Der `read`-Befehl hat die allgemeine Form:

```
read * [,v ] ...
```

wobei *v* für den Namen einer Variablen steht. Wenn der Rechner die Anweisungen des Programms bis zu einem `read`-Befehl abgearbeitet hat, wird die Ausführung des Programms unterbrochen, bis durch Drücken der ENTER-Taste am Terminal die Zeile, in der der Cursor gerade steht, als Eingabezeile abgeschickt wird. Das Programm wird erst dann fortgesetzt, wenn allen Variablen, die in der `read`-Anweisung aufgeführt sind, ein Wert zugewiesen wurde. Jeder `read`-Befehl liest vom Anfang der Zeile, in der sich der Cursor gerade befindet.

Ein Beispiel:

```
read *, a, b, c
```

Dazu die Eingabezeile auf dem Bildschirm (Das Zeichen „␣“ markiert ein Leerzeichen):

```
10␣20␣30
```

Durch Drücken der

```
ENTER-Taste          (+ENTER)
```

wird die Eingabe abgeschickt. Das Programm wird fortgesetzt und die Variablen haben die Werte:

```
a=10 b=20 c=30
```

Zwischen zwei Zahlen in einer Zeile muss ein **Trennzeichen** stehen. Mögliche Trennzeichen sind:

- Ein oder mehrere Leerzeichen: ␣
- ein Komma: `,`
- ein Schrägstrich: `/`

Das nächste Beispiel soll noch einmal die Eingabe verdeutlichen. Den Variablen müssen – entsprechend dem `read`-Befehl – Werte zugewiesen werden. Mit jedem `read`-Befehl wird eine neue Bildschirmzeile angesprochen. Zwei `read`-Befehle können nicht von einer Zeile lesen.

```
read *, x
read *, y
```

Dazu die Bildschirmzeile

```
5., 6.              (+ENTER)
```

Diese Anweisungen weisen **nicht** der Variablen *y* den Wert 6 zu. Der erste `read`-Befehl liest von der Bildschirmzeile und weist *x* den Wert 5 zu, der zweite `read`-Befehl liest von einer neuen Zeile. Die muss noch geschrieben und mit `ENTER` abgeschickt werden.

Der Stern „`*`“ in der `print *` und in der `read *` Anweisung ist ein **Formatparameter**. Fortran ist eine formatgebundene Sprache, d.h. man muss immer explizit angeben, in welchem Format die Ein- bzw. Ausgabe der Daten erfolgen soll. In den oben genannten Beispielen wurden keine Angaben über den Typ der Variablen gemacht. Real- und Integervariablen werden aber ganz unterschiedlich ein- bzw. ausgegeben. Abhängig vom Typ der Eingabe- bzw. Ausgabelistenelemente werden bestimmte intern vordefinierte Formate verwendet. Mit Hilfe des Sterns gibt man also an, dass es sich um eine **listengesteuerte Formatierung** handelt.

Ein Tip: Wenn `print`-Befehle nicht die gewünschte Ausgabe ergeben, sieht man das sofort. Wenn aber durch `read`-Befehle Variablen falsche Werte zugewiesen werden, merkt man das erst an unsinnigen Rechenresultaten. Deshalb sollte man sich unmittelbar nach der Eingabe die Werte der Variablen ausgeben lassen!

Arithmetische Ausdrücke, Numerische Operatoren:

Ab Zeile 27 tauchen Ausdrücke auf, die an mathematische Gleichungen erinnern. Es wurde aber schon gesagt, dass hier keine Aussage über eine mathematische Gleichheit vorliegt, sondern eine Zuweisung des Ausdrucks auf der rechten Seite des Gleichheitszeichens zu der Variablen auf der linken Seite.

In Zeile 27 steht auf der rechten Seite des „=“-Zeichens ein arithmetischer Ausdruck. In arithmetischen Ausdrücken werden Konstanten und/oder Variablen mittels numerischer Operatoren miteinander verknüpft. Ähnlich wie in der Mathematik können Klammern zur Festlegung der Reihenfolge der Verknüpfung verwendet werden.

Die zulässigen numerischen Operatoren sind in Tabelle 2 dargestellt.

Bei der Konstruktion von arithmetischen Ausdrücken ist darauf zu achten, dass weder zwei Operatoren noch zwei Konstanten und/oder Variablen aufeinander folgen dürfen.

+	Addition, Vorzeichen
-	Subtraktion, Vorzeichen
*	Multiplikation
/	Division
**	Exponentiation

Tabelle 2: numerische Operatoren in Fortran95

Beispiele für arithmetische Ausdrücke:

richtig:

```
3*x+2
6+12-1/3*x4**9
((3*(x+9)-N)*20)**(2*Y)
a**(-l)-1+x
```

falsch:

```
3x+2A**-1
x+-1
```

Für die Reihenfolge der Abarbeitung (**Vorrang**) gilt:

1. Klammerausdrücke (Auswertung von innen nach außen)
2. Exponentiation
3. Multiplikation und Division
4. Addition und Subtraktion

Bei **gleichrangigen** Operatoren wird von links nach rechts, nur bei der Exponentiation von rechts nach links abgearbeitet.

Bei der Auswertung von arithmetischen Ausdrücken wird vor der Auswertung zunächst geprüft, ob beide Operanden denselben Typ (Variablentyp) haben. Wenn beide Operanden den gleichen Typ haben, dann wird auch das Ergebnis von diesem Typ sein. Wenn nicht, dann werden zuerst die Operanden nach Tabelle 3 umgewandelt, bevor das Ergebnis berechnet wird (**automatische Typenumandlung**).

Operand	**Operand**	**Ergebnis**
Integer	Real	Real
Integer	Double Precision	Double Precision
Real	Double Precision	Double Precision

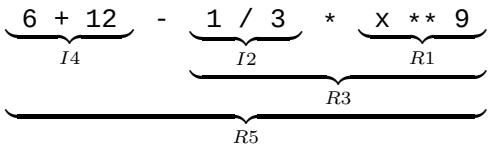
Tabelle 3: Automatische Typumwandlung arithmetischer Ausdrücke

Beispiel:

Anhand der Analyse eines arithmetischen Ausdrucks soll die Bedeutung dieser Regeln erläutert werden.

```
6 + 12   -  1 / 3   *  x ** 9
```

Reihenfolge der Bearbeitung:



Erklärung: Die höchste **Priorität** in dem arithmetischen Ausdruck hat die Potenzierung `x ** 9`. Vorausgesetzt, dass `x` vom Typ `real` ist, ist das Ergebnis der Verknüpfung eine Reale Größe, die mit *R1* bezeichnet werden soll. Die nächsthöhere Priorität haben die Division und die Multiplikation. Da bei gleicher Priorität von links nach rechts abgearbeitet wird, wird zunächst `1/3` ausgerechnet. Das Ergebnis ist die Integergröße *I2*. Nun folgt die Multiplikation von *R1* und *I2* mit dem reellen Ergebnis *R3*.
Der arithmetische Ausdruck ist bis auf die gleichberechtigte Addition und Subtraktion ausgewertet. Von links nach rechts wird nun *I4* (`6+12`) und schließlich *R5* (`I4+R3`) gebildet.

Übung:

Der analysierte Ausdruck hat für beliebige *x*-Werte immer den `real`-Wert 18! Warum?

Wie ist das bei dem scheinbar gleichwertigen Ausdruck

```
6 + 12 - x ** 9  /  3     ?
```

Programmende:

Im Programm folgt nun nur noch die `end`-Anweisung. Diese Anweisung kennzeichnet das Ende eines Programmteils. Es gibt Programme, die sich aus mehreren Teilprogrammen zusammensetzen. Jedes Teilprogramm wird durch die `end`-Anweisung abgeschlossen. Die formale `end`-Anweisung lautet:

```
end [ program name ]
```

Die Angabe des Programmteils, der beendet wird, ist optional. Bei größeren Programmen mit mehreren Unterprogrammen häufen sich die „`end`s“ sehr schnell. Die Angabe, welches Unterprogramm gerade beendet wird, hilft dann bei der Orientierung im Progammtext.

4.4 Compiler

Die grundlegenden Schritte bei der Erstellung eines ausführbaren Programms sind Gegenstand des Abschnitts 3.2 Prinzipieller Ablauf bei der Erstellung eines Programms. In diesem Abschnitt wird der *Aufruf* des Fortran95-Compilers genauer behandelt. Es wird davon ausgegangen, dass ein Programm – z.B. das Programm Henriette – bereits editiert und in einer Datei gespeichert worden ist.

Für den Compiler muss das Programm mit der Endung **`.f90`** abgespeichert werden, sonst „erkennt“ der Compiler die Datei nicht als ein Fortran95-Programm.

Die allgemeine Anweisung zum Aufruf des Compilers lautet:

```
f95 [ -Optionen ] datei
```

wobei anstelle von *datei* der Name der Datei einzusetzen ist, in der das Programm steht.

Eine Liste der möglichen Optionen bekommt man durch Aufruf des Compilers mit folgendem Parameter:

```
f95 -help
```

Das ausführbare Programm wird standardmäßig in die Datei **`a.out`** geschrieben und kann durch den Aufruf

```
a.out
```

zur Ausführung veranlasst werden. Schöner ist es natürlich, wenn auch das ausführbare Programm einen sinnvollen Namen hat. Dazu verwendet man die Option **`-o`** Mit der Anweisung:

```
f95 -o henriette henriette.f90
```

erhält das ausführbare Programm den Namen *henriette*.

4.4.1 Die Compileroption `-c`

Wie in Kapitel 3.2 erläutert, übernimmt der Compiler zwei Aufgaben: zuerst wird der eigentliche Programmtext in maschinenlesbaren Binärcode übersetzt, anschließend wird das Programm gelinkt. Die oben beschriebenen Compileraufrufe übernehmen beide Aufgaben. Wenn jedoch ein Programm nur übersetzt und nicht gelinkt werden soll, kann man die Option „`-c`" verwenden:

```
f95 -c henriette.f90
```

Hier wird zunächst nur eine **Objektdatei `henriette.o`** erzeugt, die anschließend mit der Anweisung:

```
f95 -o henriette henriette.o
```

gelinkt werden kann.

Endet der Name auf `.f` bzw. `.f90`, dann erwartet der Compiler ein Fortran Quellprogramm, endet der Name auf `.o`, dann erwartet der Compiler eine Objektdatei.

Die Möglichkeit ein Programm nur zu übersetzen und nicht zu linken, ermöglicht das Schreiben eigener **Bibliotheken**, d.h. Sammlungen von Unterprogrammen (vgl. Kapitel 6). Nachdem einmal eine Bibliothek geschrieben wurde, kann sie in eine Objektdatei übersetzt werden und muss später nur noch zu dem Programm, dass sie verwendet, hinzugelinkt werden (durch Angabe mehrerer `.o`-Dateien beim Linken). Dies erspart eine erneute Übersetzung der Bibliothek, was ggf. recht lange dauern könnte.

Bei größeren Programmprojekten ist es sehr sinnvoll das Programm in mehrere Quelldateien zu unterteilen. Diese werden getrennt übersetzt und später zusammengelinkt. Dies hat den Vorteil, dass immer nur der Teil des Programms neu übersetzt werden muss, der wirklich geändert wurde, so dass ein evtl. zweistündiger Compilerlauf auf einige Sekunden verkürzt wird. In diesem Zusammenhang ist es angebracht sog. Makefiles zu erwähnen, mit denen es möglich ist solche größeren Programmprojekte zu verwalten und sehr viel Tipparbeit bei den Compileraufrufen zu sparen. Makefiles sind allerdings nicht Thema dieses Skriptes und es bleibt dem Leser selbst überlassen, sich darüber zu informieren.

5 Kontrollstrukturen

Die bisher vorgestellten Programmkonstrukte waren rein sequentieller Natur, das heißt, die Anweisungen, aus denen die Programme bestanden, wurden in exakt vorhersagbarer Weise durchlaufen: nämlich von oben nach unten.

Soll ein Programm aber auf Benutzereingaben reagieren, oder sollen in bestimmten Situationen Programmteile alternativ oder mehrfach durchlaufen werden, ist eine Ablaufsteuerung des Programms notwendig. In solchen Fällen ist die Abfolge der Programmabarbeitung nicht mehr exakt vorhersagbar, sondern abhängig von Bedingungen, die erst zur Laufzeit ausgewertet werden. Die Klasse von Fortran95-Anweisungen, die eine solche Kontrolle ermöglicht, ist die der Kontrollstrukturen.

Beispielprogramm – Grundlegende Elemente:

Im folgenden Beispiel soll ein Programm geschrieben werden, das die Koordinaten x, y eines Punktes einliest und prüft, ob diese innerhalb eines Kreises um den Nullpunkt mit dem Radius r liegen. Falls diese Bedingung erfüllt ist, soll auf den Bildschirm ausgegeben werden: „`DER PUNKT LIEGT IM KREIS`“. Dann soll die Kreisfläche $A = \pi r^2$ berechnet und der Variablen mit dem Namen `flaeche` zugewiesen werden.

Anderenfalls soll ausgegeben werden: „`DER PUNKT LIEGT NICHT IM KREIS`“. Der Variablen `flaeche` soll in diesem Fall der Wert $\pi(x^2 + y^2 - r^2)$ zugewiesen werden. Zum Schluss soll der Wert von `flaeche` auf den Bildschirm ausgegeben werden.

Zunächst muss also eine Bedingung formuliert werden, mithilfe derer entschieden werden kann, ob der betrachtete Punkt im Kreis liegt. Die Bedingung muss lauten:

$$x^2 + y^2 < r^2$$

bzw. in Fortran95-Schreibweise:

```
x**2 + y**2 < r**2
```

5.1 Vergleichsausdrücke

Unter Fortran95 ist eine Bedingung entweder erfüllt (wahr bzw. `true`) oder nicht erfüllt (falsch bzw. `false`). Sie stellt einen Vergleichsausdruck dar. Das heißt, es werden zwei Ausdrücke (zwei Zahlen oder zwei Zeichenketten) mithilfe eines Vergleichsoperators miteinander verglichen. Die in Fortran95 zugelassenen Vergleichsoperatoren sind in Tabelle 4 aufgelistet.

Ein Vergleichsausdruck hat die Form:

```
a1 Vergleichsoperator a2
```

wobei `a1` und `a2` arithmetische Ausdrücke (Funktionen, Variablen, feste Werte etc.) oder Zeichenketten darstellen. Beispiele:

```
x < y
```

```
'Schmidt' /= 'Schmied'
```

Mit Vergleichsausdrücken werden in Fortran95 Bedingungen definiert, unter denen bestimmte Programmteile ausgeführt oder übersprungen werden. Eine Bedingung kann ein einzelner Vergleichsausdruck mit dem Ergebnis „wahr„ oder „falsch“ sein, aber auch eine Verknüpfung mehrerer Vergleichsausdrücke mit logischen Operatoren, die im Ergebnis wiederum „wahr„ oder „falsch“ sind. Beispiele für die

Vergleichsoperator (Fortran95)	mathematisches Symbol	Bedeutung
<	$<$	kleiner
<=	$\leq$	kleiner oder gleich
==	$=$	gleich
/=	$\neq$	ungleich
>	$>$	größer
>=	$\geq$	größer oder gleich

Tabelle 4: Vergleichsoperatoren in Fortran95

Verknüpfung von Vergleichsausdrücken sind in Abschnitt 12.2 Der Datentyp `logical`) angegeben.
An dieser Stelle sei nochmal ausdrücklich auf den Unterschied zwischen = und == verwiesen. Das einfache Gleichheitszeichen stellt in Fortran eine Zuweisung dar, während das doppelte Gleichheitszeichen ein Vergleichsoperator ist. Es ist deshalb unmöglich mit einem = eine Bedingung für eine Kontrollstruktur aufzustellen.

5.2 Die alternative Auswahl mit `if-else`

Zur Auswertung einer Bedingung in der Art, dass bestimmte Programmabschnitte nur bei Zutreffen der Bedingung ausgeführt werden, kommt die `if`-Blockanweisung zum Einsatz. Ihre Struktur beinhaltet neben dem Schlüsselwort `if` (*falls*) auch noch die eng dazugehörenden Schlüsselwörter `then` (*dann*) und `end if`.

Die einfachste `if`-Blockanweisung, die bedingte Auswahl, ist in Bild 10 dargestellt.

```
if (Bedingung) then
   Anweisung
end if
```

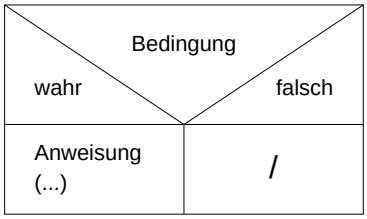


Abbildung 10: Kontrollstrukturen – Bedingte Auswahl

Die *Anweisung*[13] wird nur dann ausgeführt, wenn die *Bedingung* erfüllt ist.

Als erste Erweiterung dieses Konzepts kann eine zweite Anweisung eingefügt werden, die alternativ zur ersten ausgeführt wird. Dazu dient das Schlüsselwort `else` (*sonst*). Dieser Block wird alternative Auswahl genannt und ist in Bild 11 dargestellt.

Nun gilt: trifft die *Bedingung* zu, wird *Anweisung 1* ausgeführt, sonst *Anweisung 2*. *Anweisung 1* wird auch als `if`-Block, *Anweisung 2* als `else`-Block bezeichnet.

[13] Hier – wie auch in den folgenden Beispielen – kann ebenso mehr als eine Anweisung stehen. Die Anweisungen werden dann der Reihe nach abgearbeitet.

```
if (Bedingung) then
   Anweisung 1
else
   Anweisung 2
end if
```

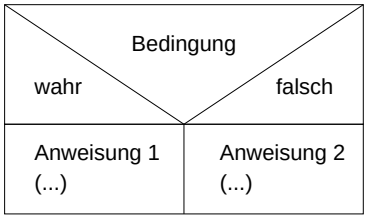


Abbildung 11: Kontrollstrukturen – Alternative Auswahl

In Bild 12 ist nun eine mögliche Lösung für das Beispielprogramm zusammen mit dem zugehörigen Struktogramm dargestellt.

```
if (x**2 + y**2 < r**2) then
   print *,'DER PUNKT LIEGT IM KREIS'
   flaeche = 3.14 * r**2
else
   print *,'DER PUNKT LIEGT NICHT IM KREIS'
   flaeche = 3.14 * (x**2 + y**2 - r**2)
end if

print *, flaeche
```

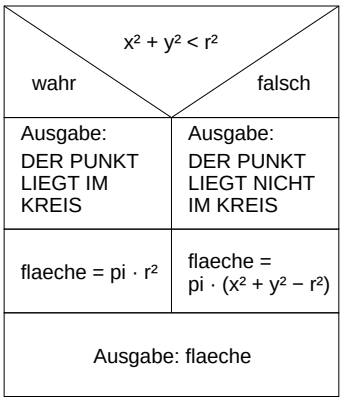


Abbildung 12: Kontrollstrukturen – Beispielprogramm Alternation

Das Programm führt entweder nur den `if`-Block oder nur den `else`-Block aus. Der im Programm folgende Befehl

```
print *, flaeche
```

wird dagegen auf jeden Fall ausgeführt, da er sich außerhalb der `if`-Blockanweisung, die mit `end if` endet, befindet.

5.3 Die bedingte Auswahl mit logischem `if`

Wenn bei der bedingten Auswahl der `if`-Block nur aus einer einzigen Anweisung besteht, kann das logische `if` benutzt werden:

```
if (Bedingung) Anweisung
```

oder als Beispiel:

```
if (x**2 + y**2 < r**2) flaeche = 3.14 * r**2
```

Das logische `if` ermöglicht also eine vereinfachte Darstellung der Abfrage; es darf nur benutzt werden, wenn die auszuführende einzelne (!) Anweisung direkt und ohne `then` nach `if (`*Bedingung*`)` in derselben Zeile folgt.

Übung:

Konstruiere mithilfe des logischen `if` einen Programmabschnitt, der den Betrag einer Zahl erzeugt und ausdruckt.

5.4 Anwendung: Genauigkeitsabfragen

In vielen numerischen Verfahren werden Zahlenfolgen durch eine Iterationsvorschrift der Art

$$x_{n+1} = f(x_n)$$

berechnet. Ist diese Folge konvergent, werden sich bei genügend großem n die Zahlen x_{n+1} und x_n nur wenig voneinander unterscheiden.

Damit ein Iterationsverfahren endlich ist, muss die Berechnung der Zahlenfolge an einem bestimmten Punkt abgebrochen werden. Dazu bietet sich als Abbruchkriterium die Frage nach der absoluten Differenz (absoluter Fehler) zweier aufeinanderfolgender Zahlen an:

$$|x_{n+1} - x_n| < ABS$$

ABS ist dabei ein problemangepasster, meist sehr kleiner Wert.

Aber aufgepasst: für

$$\begin{aligned} x_{n+1} &= 932154793 \\ x_n &= 932154792 \end{aligned}$$

wäre das Abbruchkriterium für ein gewähltes $ABS \leq 1$ nicht erfüllt, obwohl die beiden Zahlen in den ersten acht Stellen übereinstimmen. Die Frage nach dem absoluten Fehler berücksichtigt nicht die Größenordnung der Zahlen der Zahlenfolge. Man erfasst diese, indem man die absolute Differenz zweier Zahlen ins Verhältnis zur Größe einer der beiden Zahlen setzt (relativer Fehler):

$$\left| \frac{x_{n+1} - x_n}{x_n} \right| < REL$$

Betrachten wir die Wirkung der beiden Abbruchkriterien an den beiden Zahlenfolgen:

$$\begin{aligned} y_{n+1} &= \tfrac{1}{2} y_n + 1000 \qquad (\text{Grenzwert } y = 2000) \\ x_{n+1} &= \tfrac{1}{2} x_n + 0,1 \qquad (\text{Grenzwert } x = 0,2) \end{aligned}$$

mit den Schranken

$$\begin{aligned} ABS &= 0,0001 \\ REL &= 0,0001 \end{aligned}$$

Es zeigt sich, dass nur bei Abfrage nach relativen Fehlern beide Zahlenfolgen etwa in der gleichen Iterationsstufe abgebrochen werden. Bei Abfrage nach absoluten Fehlern wird die y-Folge „zu spät“, die x-Folge „zu früh“ abgebrochen.

Das nächste Beispiel zeigt aber, dass der relative Fehler bei Nullfolgen versagen kann:

$$z_{n+1} = \frac{1}{2} z_n$$

$$\lim_{n\to\infty} \frac{|z_{n+1} - z_n|}{|z_{n+1}|} = \lim_{n\to\infty} \frac{|\frac{1}{2} z_n - z_n|}{|\frac{1}{2} z_n|} = 1$$

Der relative Fehler bei dieser Folge ist konstant 1.

Bei der Berechnung von Zahlenfolgen muss also zusätzlich zur Abfrage des relativen Fehlers der Absolutwert der berechneten Zahl geprüft werden. Unterschreitet er eine vorgegebene Schranke, ist die Abbruchbedingung „relativer Fehler“ nicht sinnvoll. Beschreibung einer erweiterten Abbruchbedingung:

- Wenn der Absolutwert der Zahl einer Folge kleiner als die vorgegebene Schranke ist, dann Abbruch,
- sonst Berechnung des relativen Fehlers und Abbruch, wenn dieser die vorgegebene Schranke unterschreitet.

5.5 Die Mehrfachauswahl mit `select-case`

Mit den bislang behandelten Kontrollstrukturen konnten bis zu zwei Fälle einer Bedingung (im Endeffekt *ja* oder *nein*) unterschieden werden. Manche Probleme erfordern jedoch die Unterscheidung von mehr als zwei Fällen (z.B. *rot*, *grün* oder *gelb*), dazu kommt in Fortran95 die `case`-Anweisung zur Anwendung. Anders als im Fall von `if-else` wird bei `case` keine Bedingung ausgewertet, sondern der Inhalt einer (skalaren) Variablen oder ein (skalarer) Funktionswert untersucht.

Das Nassi-Shneiderman-Diagramm und die Fortran95-Syntax für die Mehrfachauswahl ist in Bild 13 dargestellt.

```
select case (Variable)
  case (Wert 1a[, Wert 1b] ...)
    Anweisung 1
  case (Wert 2a[, Wert 2b] ...)
    Anweisung 2
  ⋮
  case (Wert na[, Wert nb] ...)
    Anweisung n
  case default
    Anweisung n+1
end select
```

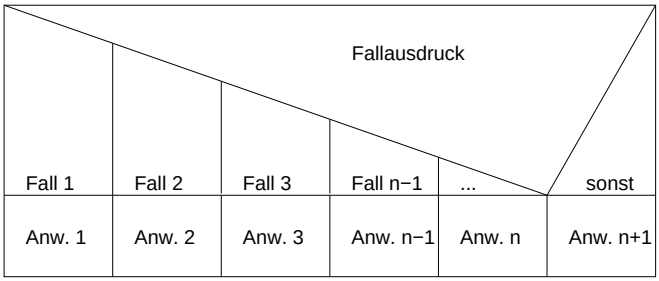


Abbildung 13: Kontrollstrukturen – Mehrfachauswahl

Die Anzahl der unterschiedenen Fälle ist beliebig, wobei sich die Falldefinitionen nicht überschneiden dürfen. Der „sonst“-Fall – auch als **`default`** bezeichnet – kann wegfallen, somit wird bei einem Durchlauf der `case`-Anweisung höchstens ein Anweisungsblock ausgeführt. Existiert der Fall `default`, so wird der dort enthaltene Anweisungsblock ausgeführt, falls kein anderer Fall zutrifft.

In der ersten Zeile des `case`-Anweisungsblocks erfolgt die Nennung der zu untersuchenden Variable. Statt einer Variable kann an dieser Stelle auch eine skalare Funktion[14] stehen. Jede Falldefinition wird

[14] Bei der Verwendung von Funktionen innerhalb der Mehrfachauswahl muss jede Funktion natürlich zum Zeitpunkt der Auswertung auswertbar sein.

dann durch das Schlüsselwort `case` eingeleitet, dem mindestens ein skalarer Wert oder eine skalare Funktion folgt. Bei der Angabe mehrerer zulässiger Werte für einen Fall werden diese durch Kommata getrennt oder – falls es sich um Zahlen oder Buchstaben handelt – ein Intervall in Doppelpunktschreibweise angegeben:

Schreibweise	Intervall	Beispiel	zugehörige Werte
`a : b`	$[a, b]$	`4 : 8`	4, 5, 6, 7, 8
		`'a' : 'd'`	'a', 'b', 'c','d'

Beispiele für gültige Falldefinitionen:

```
case (1)
case (n/2)                    ! n sei Variable
case (2, 8, 10:12, 19)        ! Zahlen 2, 8, 10, 11, 12, 19
case ('rot')
case ('der', 'die', 'das')
case ('a':'d')                ! Buchstaben a, b, c, d
case default
```

Beachtet werden muss in jedem Fall, dass der Typ der untersuchten Variable (Funktion) mit dem der in der Falldefinition angegebenen Werte (Funktionen) übereinstimmt.

5.6 Die Wiederholung mit Wiedereintrittsbedingung (`do-while`-Schleife)

Bisher haben wir nur Programme geschrieben, die linear ablaufen oder sich verzweigen. Uns fehlt noch die Möglichkeit, einen Anweisungsblock mehrfach durchlaufen zu lassen.

Beispielprogramm – Wiederholungsschleifen:

Im folgenden Beispiel soll ein Programm geschrieben werden, das ganze Zahlen dividiert, ohne den Divisionsoperator zu benutzen, und das Ergebnis als ganze Zahl ohne den Rest ausgibt. Dazu wird eine Variable `divisor` benötigt, die die Zahl enthält, die geteilt werden soll, eine Variable `dividend` für den Teiler und eine Variable `quotient` für das Ergebnis.

Vorgegangen wird folgendermaßen: solange der Divisor größer oder gleich dem Dividend ist, wird der Dividend (wiederholt) vom Divisor abgezogen. Die Anzahl der möglichen Wiederholungen ist das Ergebnis der Division ohne Rest.

In Bild 14 ist dieses Beispielprogramm zusammen mit dem zugehörigen Struktogramm dargestellt.

Drei neue Fortran95-Elemente treten hier auf: die Schlüsselwörter `do` (*tue*, im Sinne von *wiederhole*), `while` (*solange*) und `end do`. Das Schlüsselwort `do` leitet eine allgemeine Wiederholungs-Blockanweisung ein, `while` legt fest, dass jeder Schleifendurchlauf an eine Bedingung geknüpft ist: nur solange diese erfüllt ist, wird die Schleife abgearbeitet. Die vorliegende Schleife wird deshalb als Wiederholung mit **Wiedereintrittsbedingung** bezeichnet. Schließlich sorgt das Schlüsselwort `end do` analog zum oben eingeführten `end if` für eine Kapselung dessen, was innerhalb der Schleife passieren soll.

Die allgemeine Darstellung der Wiederholung mit Wiedereintrittsbedingung ist in Bild 15 zu sehen.

Wie oben bereits erwähnt, sorgt `while` dafür, dass vor jedem Schleifendurchlauf die Bedingung überprüft wird. Nur wenn sie erfüllt ist, wird die Anweisung (der Block von Anweisungen) abgearbeitet. Nach

```
integer :: quotient, divisor, dividend

quotient = 0

do while (divisor >= dividend)
   divisor = divisor - dividend
   quotient = quotient + 1
end do

print *, quotient
```

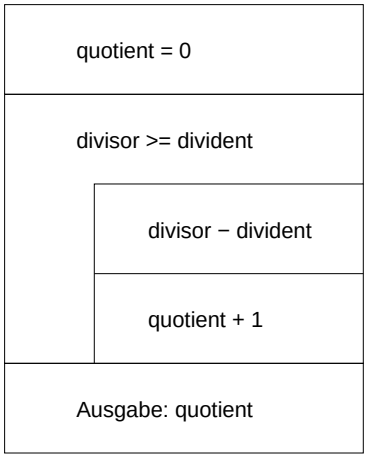


Abbildung 14: Kontrollstrukturen – Beispielprogramm Wiederholung

```
do while (Bedingung)
   Anweisung
end do
```

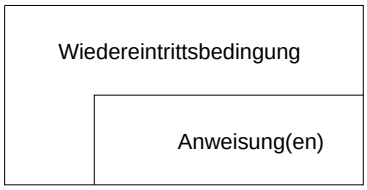


Abbildung 15: Kontrollstrukturen – Wiederholung kopfgesteuert

der Ausführung der letzten Anweisung vor dem `end do` wird dann erneut die Bedingung geprüft. Sollte die Bedingung während des Schleifendurchlaufs (also zwischen zwei Anweisungen des Anweisungsblocks) unwahr werden, wird die Schleife dennoch erst nach der Ausführung des gesamten Anweisungsblocks beendet.

Die vorgestellte Form der Schleife mit Wiedereintrittsbedingung nennt sich auch „**kopfgesteuerte Schleife**“, da die Überprüfung der Wiedereintrittsbedingung bereits im Schleifenkopf und somit noch vor dem ersten Schleifendurchlauf stattfindet. Im Gegensatz dazu wird bei der „**fußgesteuerten Schleife**“ in jedem Fall einmal die gesamte Schleife durchlaufen und erst dann geprüft, ob ein weiterer Durchlauf erfolgen soll.

In Fortran95 ist eine solche fußgesteuerte Schleife nicht explizit vorgesehen. Dennoch wollen wir hier einen Weg vorstellen, der die Implementierung ermöglicht. Dazu muss zunächst die sogenannte „Endlosschleife“ eingeführt werden, die einen Spezialfall der bereits bekannten `do-while`-Schleife darstellt.

Während bei der `do-while`-Schleife die auf das Schlüsselwort `while` folgende Bedingung für den kontrollierten Abbruch der Schleife sorgte, fehlen sowohl `while` als auch die Bedingung bei der Endlosschleife – dadurch terminiert diese Art der Wiederholungsanweisung nicht automatisch. Deshalb werden wir mithilfe einer bedingten `if`-Auswahl (siehe Abschnitt 5.3 Die bedingte Auswahl mit logischem `if`) den Schleifenabbruch „von Hand“ sicherstellen.

Die allgemeine Form einer **Endlosschleife** ist zunächst folgende:

```
do
   Anweisung
end do
```

Nun muss die `if`-Anweisung, die für den Schleifenabbruch sorgt, noch an geeigneter Stelle in die Schleife eingefügt werden. Damit aus der Endlosschleife eine fußgesteuerte Schleife wird, muss die Überprüfung der *Bedingung* als letzte Anweisung innerhalb der Schleife geschehen. Der Befehl **`exit`** sorgt hier bei *zutreffender Bedingung* für einen *Abbruch der Schleife* (!) – im Gegensatz zu den bisher verwendeten Wiedereintrittsbedingungen, die bei Zutreffen einen erneuten Schleifendurchlauf einleiteten. Die Implementierung einer fußgesteuerten Schleife in Fortran95 ist in Bild 16 dargestellt.

```
do
   Anweisung
   if (Bedingung) exit
end do
```

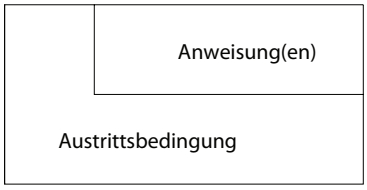


Abbildung 16: Kontrollstrukturen – Wiederholung fußgesteuert

Übung:

Die folgende Summe soll gebildet werden (Darstellung in Struktogramm und Fortran95):

$$\sum_{k=1}^{5} 2^{-k}$$

Durch Nachvollziehen des Entwurfes soll die Richtigkeit geprüft werden.

5.7 Die Wiederholung mit Zählvorschrift (do-Zählschleife)

Wenn bei einem Anweisungsblock, der mehrfach durchlaufen werden soll, die Anzahl der erforderlichen Wiederholungen bekannt ist, benutzt man gerne die sogenannte *Zählschleife*. Statt einer Wiedereintrittsbedingung enthält sie eine Zählvorschrift, bestehend aus einer Zählvariable, die bei jedem Schleifendurchlauf automatisch herauf- bzw. herabgezählt wird, einer Schranke und einer Schrittweite.

Zählschleifen werden in Fortran95 ebenfalls mit der `do`-Anweisung verwirklicht. Die allgemeine Form ist folgende:

```
do Zählvariable = Anfangswert, Schranke...
                  ...[, Schrittweite] ...
   Anweisung
end do
```

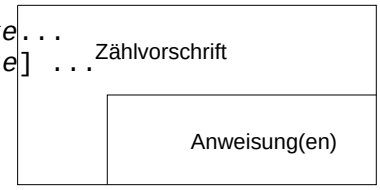


Abbildung 17: Kontrollstrukturen – Zählschleife

Im Gegensatz zu der oben eingeführten `while`-Schleife entfällt hier das Schlüsselwort `while`, an seine Stelle tritt die Zählvorschrift. Sie beinhaltet die Initialisierung der Zählvariablen mit dem *Anfangswert*, de-

finiert eine *Schranke* für den Wert der Zählvariablen (entweder den größten oder den kleinsten zulässigen Wert – je nach Schrittrichtung) und legt schließlich die *Schrittweite* der Erhöhung bzw. Erniedrigung der Zählvariablen nach jedem Schleifendurchlauf fest.

Beispielprogramm – Zählschleifen:

Angenommen, wir wollen ein Programm schreiben, das die ungeraden ganzen Zahlen von 1 bis `anzahl` aufsummiert; die Anzahl der Summationen ist also vor Eintritt in die Zählschleife bekannt.

Die Zählvariable (hier `zaehler` genannt) läuft in unserem Beispiel (Bild 18) vom Anfangswert (hier mit dem Wert 1) bis zur Schranke (hier `anzahl` genannt) mit einer bestimmten Schrittweite (hier mit dem Wert 2, weil nur die ungeraden Zahlen interessieren).

```
summe = 0

do zaehler = 1, anzahl, 2
   summe = summe + zaehler
end do
```

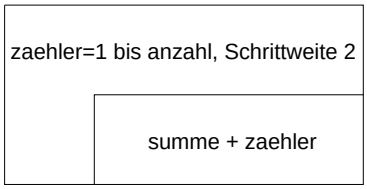


Abbildung 18: Kontrollstrukturen – Beispiel einer Zählschleife

Für `anzahl=7` ergibt sich folgender Programmablauf:

	`zaehler`	`summe`
vor Erreichen der Schleife Initialisierung von `summe`		0
Schleifenkopf erreicht Initialisierung von `zaehler`	1	0
erster Schleifendurchlauf	1	0+1=1
zweiter Schleifendurchlauf	3	1+3=4
dritter Schleifendurchlauf	5	4+5=9
vierter Schleifendurchlauf	7	9+7=16
Schranke (`anzahl=7`) überschritten Schleife beendet	9	16

Wäre als Schranke `anzahl=8` angegeben worden, wäre das Programm auf die gleiche Art und Weise abgelaufen, da `zaehler` nach dem vierten Schleifendurchlauf nicht noch einmal um 2 hätte erhöht werden dürfen (also auf den Wert 9), ohne die Schranke zu überschreiten.

Überschreitet der Wert der Zählvariablen die Schranke schon bei der Initialisierung, wird die Schleife gar nicht durchlaufen. Wird die Schranke innerhalb der Schleife durch eine Wertzuweisung geändert, so hat das keinen Einfluss auf die Anzahl der Schleifendurchläufe, da schon beim ersten Erreichen des Schleifenkopfes die Anzahl der Durchläufe berechnet wird. Die Zählvariable darf innerhalb des Schleifenrumpfes nicht verändert werden.

Wird im Schleifenkopf eine negative Schrittweite angegeben, zählt die Zählvariable rückwärts.

Die Ausführung einer `do`-Anweisung bewirkt folgende Operationen:

1. Die numerischen Ausdrücke *Anfangswert*, *Schranke*, *Schrittweite* werden berechnet und gegebenenfalls in den Typ der Zählvariablen umgewandelt.
2. Die Zählvariable `i` wird mit dem *Anfangswert* besetzt.
3. Die Anzahl der Schleifendurchläufe wird bestimmt aus der Gleichung:

$$d = \max\left(integer\left(\frac{Schranke - Anfangswert + Schrittweite}{Schrittweite}\right), 0\right)$$

4. Falls d=0 ist, wird die do-Schleife nicht durchlaufen, und das Programm wird mit der Anweisung, die dem abschließenden `end do` folgt, fortgesetzt. Der Wert der Zählvariablen entspricht in diesem Fall dem Anfangswert.

   Falls d ungleich Null ist, wird die do-Schleife d mal durchlaufen. Dabei wird nach jedem Schleifendurchlauf (auch nach dem letzten) die Zählvariable um *Schrittweite* erhöht.

Übungen:

- Welchen Inhalt besitzt `sum` nach Beendigung der Anweisung?

```
sum = 0
do alpha = 1, 5, 1
   sum = sum + alpha ** 2
end do
```

- Für welche Werte der Laufvariablen `i` wird folgende `do`-Anweisung durchlaufen?

```
do i = 1, 6, 2
```

- Es soll die Fakultät einer natürlichen Zahl berechnet werden (Struktogramm und Fortran95-Schreibweise). Zur Erinnerung:

$$\begin{aligned} n! &= 1 * 2 * \ldots * n \qquad \text{für } n \in I\!N,\ n \geq 1 \\ 0! &= 1 \end{aligned}$$

6 Prozedurale Programmierung und Unterprogramme

Bei **prozeduraler Programmierung** handelt es sich um eine Programmiermethode, bei der das zu lösende Problem zunächst in einfache Teilprobleme zerlegt wird und dann deren Lösungen wieder zu einer Gesamtlösung zusammengesetzt werden (Teilprobleme können natürlich in weitere noch einfachere „Unterteilprobleme“ zerlegt werden). Die meisten modernen Programmiersprachen (auch Fortran95) bauen auf diesem Konzept auf und stellen dem Programmierer entsprechende Hilfsmittel zur Verfügung. Das verbreitetste dieser Hilfsmittel ist das **Unterprogramm**.

6.1 Unterprogramme

Ein Unterprogramm[15] ist, wie der Name schon sagt, ein nahezu eigenständiges kleines Programm, dass vom **Hauptprogramm** oder von einem anderen Unterprogramm aufgerufen wird und eine bestimmte Aufgabe erledigt (also ein Teilproblem löst). Informationen, die das Unterprogramm für die Bewältigung seiner Aufgabe benötigt, können ihm bei seinem Aufruf in Form von **Argumenten** (oder in Fortran-Terminologie: in Form von **Aktualparametern**) mitgegeben werden. Ebenso können eventuelle Ergebnisse des Unterprogrammes in Form von **Rückgabewerten** an das Hauptprogramm oder das aufrufende Unterprogramm zurückgegeben werden. Fortran95 definiert selber eine ganze Reihe eingebauter Unterprogramme, die alltägliche Programmieraufgaben lösen, also z.B. den Sinus von einem Winkel berechnen oder zwei Matrizen multiplizieren, aber dazu später mehr (vgl. Kapitel 6.6).

Die Verwendung vom Unterprogrammen bietet einige ganz erhebliche Vorteile:

- Aufgaben, die ein Programm mehr als einmal erledigen muss (z.B. das multiplizieren von Matrizen), müssen nur einmal geschrieben werden und können dann beliebig oft verwendet werden.
- Die Verwendung von vielen Unterprogrammen mit **sinnvollen** und **aussagekräftigen Namen** kann die Lesbarkeit (und somit die Verständlichkeit) des Programmtextes extrem erhöhen.
- Beim groben Programmentwurf müssen nur die benötigten Unterprogramme, sowieso deren **Parameterlisten** (vgl. Kapitel 6.2) festgelegt werden. Die dadurch festgelegten Schnittstellen ermöglichen das Aufteilen der eigentlichen Programmierarbeit auf mehrere Personen.
- Thematisch zusammenhängende Unterprogramme können zu sog. **Bibliotheken** zusammengefasst werden, die anderen Programmierern zugänglich gemacht werden können, so dass diese das „Rad nicht neu erfinden” müssen.

6.2 Unterprogramme in Fortran

In Fortran gibt es zwei verschiedene Typen von Unterprogrammen: **functions** und **subroutines**. Beide unterscheiden sich in der Art, wie sie aufgerufen werden und in der Art, wie sie Ergebnisse an das Hauptprogramm zurückgeben.

[15] als Synonym für Unterprogramme werden häufig auch folgende Ausdrücke verwendet: Prozedur (procedure), Unterroutine (subroutine), Funktion (function)

6.2.1 subroutine-Unterprogramme

Hier zunächst ein Beispiel, wie eine Subroutine definiert und aufgerufen wird:

```
! Beispielprogramm zur Demonstration von Subroutines


! Hauptprogramm

program rufeSubroutine
implicit none

! Interfaces aller Unterprogramme, die vom Hauptprogramm
! aufgerufen werden
interface
    subroutine machWas
    implicit none
    end subroutine machWas
end interface

    ! Subroutine aufrufen
    call machWas

end program rufeSubRoutine



! Unterprogramm 'machWas' in Form einer Subroutine

subroutine machWas
implicit none
integer :: ergebnis

    ! Mache irgendwas
    ergebnis = 1 + 2
    print *, 'Das Ergebnis ist:', ergebnis

end subroutine machWas
```

Dieses Beispielprogramm gibt `Das Ergebnis ist: 3` auf dem Bildschirm aus. Es ist in zwei Teile gegliedert: das Hauptprogramm oben und die Subroutine `machWas` unten.

Beginnen wir bei der Subroutine: wie leicht zu erkennen ist, unterscheidet sich die Definition einer Subroutine nur durch das Schlüsselwort `subroutine` von der Definition des Hauptprogramms, alles andere kann völlig analog übernommen werden (d.h `implicit none`, der Variablen-Deklarationsteil und der Programmtextteil).

Im Hauptprogramm (bzw. im aufrufenden Unterprogramm) tauchen einige weitere neue Elemente auf: zunächst müssen alle Unterprogramme, die vom Hauptprogramm aufgerufen werden, diesem bekannt gemacht werden. Dies geschieht mit Hilfe eines `interface`-Blockes. Hier werden die Deklarationsköpfe[16] aller verwendeten Unterprogramme eingetragen. Um im Programmverlauf dann eine Subroutine aufzurufen, wird die `call`-Anweisung verwendet.

[16] der Deklarationskopf besteht aus der `subroutine`–Zeile, `implicit none`, der Deklaration aller Variablen der Parameterliste und der `end subroutine`–Zeile

Die gerade beschriebene Subroutine ist das einfachste Beispiel eines Unterprogramms in Fortran, das in der Praxis nicht so häufig vorkommen wird. Im Allgemeinen wird man einer Subroutine zusätzliche Informationen mitgeben wollen, die diese benötigt, um ihre Aufgabe zu erfüllen. Dies geschieht in Form von sog. **Parameterlisten**. Die Funktionsweise von Parameterlisten kann am besten anhand eines Beispiels erklärt werden:

```
! 2. Beispielprogramm zur Demonstration von Subroutines

! Hauptprogramm

program rufeSubroutine
implicit none

! Deklarationsteil
real    :: a, b, summe, ergebnis

! Interfaces aller verwendeten Unterprogramme
interface
    subroutine addiere( summe, summand1, summand2 )
    implicit none
    real, intent( out ) :: summe
    real, intent( in )  :: summand1, summand2
    end subroutine addiere
end interface

    ! die folgenden vier Aufrufe von 'addiere' sind gleichwertig
    ! nach dem jeweiligen Aufruf beinhaltet 'summe', bzw.
    ! 'ergebnis' das Ergebnis der Rechnung.
    call addiere( summe, 1.0, 2.0 )
    a = 1.0
    b = 2.0
    call addiere( summe, a, b )
    call addiere( ergebnis, a, b )
    call addiere( ergebnis, a, a + a )

end program rufeSubRoutine


! Subroutine 'addiere': addiert zwei Zahlen

subroutine addiere( summe, summand1, summand2 )
implicit none
real, intent( out ) :: summe
real, intent( in )  :: summand1, summand2

    ! addiere beide Zahlen
    summe = summand1 + summand2

end subroutine addiere
```

Wenden wir wiederum zunächst den Blick der Subroutine zu: hinter dem Namen der Subroutine wird in Klammern ihre Parameterliste aufgeführt. Diese besteht aus mit Kommata getrennten Variablennamen (**Parameter** bzw. in Fortran-Terminologie: **Formalparameter**), die zur Kommunikation mit der „Außenwelt" dienen. Diese Variablen müssen im folgenden Deklararationsteil definiert werden. Dabei wird zwi-

schen Eingabe- und Ausgabevariablen unterschieden, die jeweils das Attribut `intent( in )` bzw. `intent( out )` tragen sollten. Im vorliegenden Beispiel gibt es also zwei Eingabevariablen ‚summand1' und ‚summand2', deren Inhalt beim Aufruf der Subroutine durch die übergebenen Argumente festgelegt werden, und eine Ausgabevariable, die den errechneten Wert wieder an das Hauptprogramm (oder das aufrufende Unterprogramm) zurückliefert. Es ist auch möglich Variablen zu definieren, die sowohl der Ein-, als auch der Ausgabe dienen, diese tragen das Attribut `intent( inout )`.

Achtung!
Alle Variablen einer Parameterliste werden an das aufgerufene Unterprogramm übergeben und dieses gibt sie auch sämtlich wieder an das aufrufende Programmteil zurück, unabhängig von den `intent`-Attributen die für diese Variablen definiert wurden.
Die `intent`-Attribute sind eher wie Lese- oder Schreibrechte auf Variablen zu interpretieren. Das `intent( in )`-Attribut legt fest, dass das Unterprogramm welches diese Variable übergeben bekommt diese nur lesen darf und auf keine Fall beschreiben (wenn man das trotzdem versucht gibt der Compiler eine Fehlermeldung aus). Das `intent( out )`-Attribut legt fest, dass das Unterprogramm diese Variable beschreiben darf und soll (wenn man dieser Variablen keinen Wert zuweist, sie also nicht beschreibt innerhalb des Unterprogrammes, so gibt der Compiler eine Fehlermeldung aus). Das `intent( inout )`-Attribut gewährt dem Unterprogramm freien Zugriff auf die Variable.

Um diese Subroutine im Hauptprogramm aufrufen zu können, müssen zunächst die Definitionen der Variablen der Parameterliste in den `interface`-Block übernommen werden. Der Aufruf selber geschieht dann, indem hinter dem Namen der Subroutine in Klammern deren Argumente (bzw. Aktualparameter) angegeben werden. Der Wert eines Argumentes wird vom entsprechenden Parameter der Subroutine übernommen, also an die **Subroutine übergeben** (Die Zuordnung von Argumenten zu Parametern erfolgt dabei streng nach deren **Reihenfolge** in der Parameterliste). Argumente können entweder Konstanten, Variablen oder komplette Rechenausdrücke sein, die ausgewertet werden, bevor die Subroutine aufgerufen wird. Im Falle, dass Variablen übergeben werden, spielt deren Name keine Rolle, sie können im Hauptprogramm entweder den gleichen oder auch einen anderen Namen tragen, als deren Pendants im Unterprogramm. Wenn eine Subroutine, wie in diesem Beispiel, über die Parameterliste Werte an den aufrufenden Programmteil zurückgibt, müssen die entsprechenden Argumente natürlich aus Variablen bestehen, um diese Werte aufnehmen zu können.

Allgemein wird eine Subroutine so definiert:

```
subroutine name [(fp[, fp] ...)]
implicit none

! Vereinbarung der Formalparameter und weiterer Variablen
typ, intent( in/out/inout ) :: fp [, fp] ...
typ :: variablennamen

! Programmtext
⋮
end [subroutine name]
```

Mit dem dazugehörigen Aufruf:

```
call name [(ap [, ap] ...)]
```

6.2.2 `function`-Unterprogramme

`function`-Unterprogramme sind `subroutine`-Unterprogrammen sehr ähnlich. Sie werden mit dem `function`-Schlüsselwort eingeleitet und verfügen genau wie Subroutinen über eine Parameterliste, zusätzlich besitzen sie jedoch noch einen von der Parameterliste unabhängigen Rückgabewert, wie das folgende Beispiel demonstriert:

```
! Beispielprogramm zur Demonstration von Functions

! Hauptprogramm

program rufeFunction
implicit none

! Deklarationsteil
real    :: a, b, summe, ergebnis

! Interfaces aller verwendeten Unterprogramme
interface
    function addiere( summand1, summand2 )
    implicit none
    real :: addiere
    real, intent( in )  :: summand1, summand2
    end function addiere
end interface

    ! die folgenden vier Aufrufe von 'addiere' sind gleichwertig
    ! nach dem jeweiligen Aufruf beinhaltet 'summe', bzw.
    ! 'ergebnis' das Ergebnis der Rechnung.
    summe = addiere( 1.0, 2.0 )
    a = 1.0
    b = 2.0
    summe = addiere( a, b )
    ergebnis = addiere( a, b )
    ergebnis = addiere( a, addiere( a, a ) )

end program rufeFunction



! Function 'addiere': addiert zwei Zahlen

function addiere( summand1, summand2 )
implicit none
real :: addiere
real, intent( in )  :: summand1, summand2

    ! addiere beide Zahlen
    addiere = summand1 + summand2

end function addiere
```

Dieses Beispielprogramm bewältigt die gleiche Aufgabe, wie das vorige Beispielprogramm zum Thema Subroutinen. Der einzige Unterschied besteht darin, dass hier `addiere` keine `subroutine`, sondern eine `function` ist, und die Rückgabe des Ergebnisses nicht über die Parameterliste erfolgt, sondern über einen Rückgabewert. Dieser Rückgabewert wird von der Funktion gesetzt, indem der Variablen, die den gleichen Namen trägt, wie die `function` selber, ein Wert zugewiesen wird. Diese Variable muss wie alle anderen Variablen auch im Deklarationsteil der `function` definiert werden. Hier darf allerdings kein `intent( out )`-Attribut angegeben werden, da diese Variable automatisch eine reine Ausgabevariable ist. Innerhalb der `function` darf daher auch nicht lesend auf diese Variable zugegriffen werden, folgendes Beispiel funktioniert deshalb nicht:

```
function gehtNicht( irgendWas )
implicit none
integer                 :: gehtNicht
integer, intent( in )   :: irgendWas

    ! hier wird lesend auf die Rueckgabevariable zugegriffen,
    ! daher verursacht folgende Zeile eine Fehlermeldung!
    gehtNicht = gehtNicht + irgendWas;

end function gehtNicht
```

Der Aufruf einer `function` erfolgt ohne die `call`-Anweisung, einfach indem der `function`-Name und die Argumente in Klammern angegeben werden. Allerdings muss der Rückgabewert auf jeden Fall ausgewertet werden, d.h. in einer Rechnung weiterverarbeitet oder einer Variablen zugewiesen werden. Anschaulich kann man sich vorstellen, dass während des Programmablaufes der Aufruf einer `function` gegen deren Ergebnis (Rückgabewert) ersetzt wird.

Rein formal sieht die Definition einer `function` so aus:

```
function name ([fp[, fp] ...])
implicit none

! Vereinbarung der Formalparameter, des Rueckgabewertes
! und weiterer Variablen
typ :: name
typ, intent( in/out/inout ) :: fp [, fp] ...
typ :: variablennamen

! Programmtext der Funktion
⋮

! Zuweisung des Rueckgabewertes
name = ...
⋮

end [function name]
```

Mit diesem Aufruf:

```
variable = name ([ap [, ap] ...])
```

Anmerkungen:

- Eine `function` darf, genau wie eine `subroutine`, eine leere Parameterliste besitzen, allerdings muss im Gegensatz zur Subroutine dann ein leeres Klammerpaar sowohl in der `function`-Definition, als auch beim `function`-Aufruf angegeben werden. Ein Weglassen der Klammern ist nicht erlaubt.
- Genau wie eine Subroutine kann eine `function` **zusätzlich** zu ihrem normalen Rückgabewert **weitere Werte über die Parameterliste zurückgeben**, deshalb sollte auch hier immer mit `intent( ... )` gearbeitet werden.
- Es gibt auch eine alternative Form den Rückgabewert einer `function` zu definieren: statt ihn im Deklarationsteil aufzuführen, kann der Typ des Rückgabewertes vor dem `function`-Schlüsselwort angegeben werden.

  Beispiel:

  ```
  real function addiere( summand1, summand2 )
  implicit none
  real, intent( in ) :: summand1, summand2

      addiere = summand1 + summand2

  end function addiere
  ```

  Allerdings ist es so nur möglich einfache Datentypen als Rückgabewerte zu verwenden, bei Feldern (vgl. Kapitel 7) zum Beispiel versagt diese Methode.

6.2.3 `function` vs. `subroutine`

Jetzt wissen wir, dass Fortran zwei verschiedene Möglichkeiten zur Verfügung stellt, Unterprogramme zu definieren. Aber für welche der beiden Möglichkeiten sollte man sich in der Praxis entscheiden? Die Antwort lässt sich natürlich nicht pauschal geben und ist oft eine Frage des „Geschmacks".

Wenn Unterprogramme nur einen Rückgabewert besitzen, ist es mit Sicherheit etwas eleganter eine `function` zu verwenden, vor allem da man `function`-Aufrufe schachteln kann.

Beispiel:

Angenommen es gibt drei `functions`: min, max und machWas die alle drei 2 Übergabeparameter erwarten, und drei Variablen: a, b, ergebnis (der Typ ist hier unwichtig), dann lassen sich die Aufrufe dieser `functions` wie folgt kombinieren:

```
ergebnis = machWas( min( a, b ), max( a, b ) )
```

Insbesondere bei mathematischen Funktionen ist das sehr praktisch, da man so die Syntax aus der Mathematik (zumindest teilweise) übernehmen kann.

Wenn ein Unterprogramm keinen Rückgabewert besitzt, verwendet man natürlich eine `subroutine`. Allerdings macht es Sinn bei vielen Unterprogrammen, die von ihrer eigentlichen Aufgabe her keinen Rückgabewert besitzen, einen Rückgabewert als Status- oder Fehlermeldung einzuführen. So benötigt z.B. ein Unterprogramm, das Daten in eine Datei schreibt, zunächst einmal keinen Rückgabewert. Allerdings könnte es ja vorkommen, dass es auf dem Datenträger keinen Platz mehr für diese Datei gibt und die Operation fehlschlägt. In diesem Fall sollte der aufrufende Programmteil (also z.B. das Hauptprogramm) von diesem Fehlschlag informiert werden – in Form eines Rückgabewertes, also über eine `function`. In der Praxis gibt es daher nur wenige Unterprogramme, die keinen Rückgabewert benötigen.

Ein Unterprogramm, das mehrere Rückgabewerte erzeugt, wird man meist als `subroutine` ausführen, da auf diese Weise kein Rückgabewert dem anderen gegenüber „bevorzugt" behandelt wird, was evtl. sonst nicht der Logik des Programms entsprechen würde.

Zusammenfassend lässt sich sagen, dass die Fälle in denen man `functions` einsetzen wird i.A. häufiger auftreten, als jene, in denen man `subroutinen` einsetzt.

6.3 Module

Dem einen oder anderen Leser wird vielleicht aufgefallen sein, dass die Verwendung von Interfaceblöcken leicht in viel Arbeit ausarten kann: ein kurzes Unterprogramm, das auf mehrere weitere Unterprogramme zurückgreift, besitzt leicht einen Interfaceblock, der um ein Vielfaches länger ist, als sein eigentlicher Programmtext. In größeren Programmen ist der damit verbundene Aufwand nicht mehr zu

rechtfertigen. In solchen Fällen können **Module** verwendet werden. Unser Beispielprogramm sieht nach dem Einsatz von Modulen so aus:

```
! Beispielprogramm zur Demonstration von Modules


! Modul mit der Funktionsdefinition

module Mathe
contains

    function addiere( summand1, summand2 )
    implicit none
    real                :: addiere
    real, intent( in )  :: summand1, summand2

        addiere = summand1 + summand2

    end function addiere

end module Mathe


! Hauptprogramm

program rufeFunction
use Mathe
implicit none

! Deklarationsteil
real    ::  summe

    summe = addiere( 1.0, 2.0 )

end program rufeFunction
```

Die `function`-Definition befindet sich jetzt innerhalb des Modules `Mathe`. Ähnlich wie das Hauptprogramm oder Unterprogramme wird ein Modul mit dem Schlüsselwort `module` gefolgt vom Modulnamen eingeleitet und endet mit `end module` *name*. Innerhalb eines Modules können Variablen, Konstanten, Unterprogramme u.ä. definiert werden (vgl. auch Kapitel 6.4), die dann auf einen Schlag über ein simples `use` *name* dem Hauptprogramm oder einem beliebigen Unterprogramm zugänglich gemacht werden können. Die Verwendung von Interfaces entfällt in diesem Fall. Innerhalb eines Modules können natürlich mehrere Unterprogramme definiert werden, die sich auch gegenseitig aufrufen können – in einem solchen Fall ist weder ein Interface noch eine `use`-Anweisung notwendig.

Die formale Definition eines Modules sieht so aus:

```
module name

    ! Typdefinitionen und Variablendeklarationen
    ⋮

contains

    ! Definition von Unterprogrammen
    ⋮

end module name
```

6.4 Lokale und globale Variablen

Die meisten Programmiersprachen kennen, im Gegensatz zu Fortran, das Konzept von lokalen und globalen Variablen.

Lokale Variablen sind nur innerhalb des Hauptprogramms oder innerhalb eines Unterprogramms bekannt. Wenn ein anderes Unterprogramm aufgerufen wird, bringt dies einen komplett neuen Satz von lokalen Variablen mit und kann nicht auf die Variablen des aufrufenden Programmteils zugreifen (es sei denn sie wurden über die Parameterliste übergeben). Daher ist es möglich den gleichen Variablennamen in verschiedenen Unterprogrammen zu verwenden, es handelt sich dann um verschiedene Variablen, die alle den gleichen Namen besitzen, von denen aber immer nur gerade eine „aktiv“ ist. Im Gegensatz dazu sind globale Variablen in jedem Unterprogramm bekannt und können von überall verwendet werden.

In diesem Sinne sind also alle Variablen in Fortran lokale Variablen. Es gibt jedoch verschiedene Möglichkeiten, globale Variablen zu „simulieren”. Hier soll nur auf eine dieser Möglichkeiten eingegangen werden, da die anderen recht schwer zu handhaben und fehleranfällig sind. Bei dieser Möglichkeit handelt es sich wiederum um Module.

Variablen, die in Modulen definiert werden, können von allen Programmteilen, die diese Module verwenden, ausgewertet und verändert werden, man könnte sagen, diese Variablen sind pseudo-global (da sie ja nur in einigen und nicht in allen Unterprogrammen bekannt sind).

Vorsicht! Globale Variablen sollten nur in wenigen Ausnahmefällen verwendet werden, denn sie tragen in hohem Maße dazu bei ein Programm schwer verständlich und anfällig für die verschiedensten Arten von Fehlern zu machen. Insbesondere jede Art von Zählvariablen oder Hilfsvariablen sollte auf gar keinen Fall „globalisiert“ werden.

Beispiel:

```
module Global
    integer :: i
end module Global


program Globalisierungsgefahr
use Global
implicit none
interface
    subroutine machWas
    implicit none
    end subroutine machwas
end interface

    ! hier wird das globale i verwendet
    do i=1,10
        call machWas
    end do

end program Globalisierungsgefahr


subroutine machWas
use Global
implicit none

    ! und hier wird nochmal das globale i verwendet
    ! => KONFLIKT!
    do i=1,5
        print *,i
    end do

end subroutine machWas
```

Dieses Programm besteht aus einer Endlosschleife (d.h. dieses Programm wird nie enden)! Warum? – Im Hauptprogramm zählt eine Schleife mit der globalen Zählvariablen `i` von eins bis zehn. Nach dem Eintritt in die Schleife hat `i` also den Wert 1. In dem Unterprogramm, das dann aufgerufen wird, wird wieder die selbe Variable als Zählvariable benutzt und hat den Wert 5, wenn das Unterprogramm beendet wird. Im Hauptprogramm hat `i` dann also auch den Wert 5. Im nächsten Schleifendurchlauf wird dann `i` auf 6 erhöht, aber nach dem Aufruf von `machWas`, hat es wieder den Wert 5. Und dieser Vorgang wiederholt sich bis ins Endlose.

Einer der wenigen Fälle, in denen eine globale Variable sinnvoll wäre, ist eine, wie auch immer geartete, Datenbankanwendung. Hier könnte die eigentliche Datenbank global gestalten werden, da ja (fast) jedes Unterprogramm auf diese Datenbank zugreifen wird. Die Alternative wäre eine lokale Datenbank im Hauptprogramm, die dann an alle Unterprogramme als Argumente „durchgereicht“ wird. In diesem Falle wäre die Verwendung einer globalen Datenbank mit Sicherheit eleganter, aber nicht zwingend notwendig.

Bevor man also in der Praxis eine globale Variable einsetzt, sollte man gründlich prüfen, ob dies wirklich erforderlich ist und ggf. davon abstand nehmen.

6.5 Weitere Möglichkeiten von Unterprogrammen

Einige Möglichkeiten, die Unterprogramme bieten, wurden bisher nicht erwähnt, dies soll hier nachgeholt werden.

6.5.1 Die `return`-Anweisung

Ein Unterprogramm kann mit Hilfe der `return`-Anweisung an beliebiger Stelle unterbrochen werden. In der Praxis setzt man dies oft im Zusammenhang mit einer Fallunterscheidung ein.

Beispiel:

Diese Funktion möchte den Inhalt des Parameters `wert` in die Datei mit dem Namen, der in `datei` steht, schreiben. Beim Öffnen der Datei kann es zu Problemen kommen, dann gibt die Funktion `.false.` zurück, ansonsten `.true.`. Datei Ein- und Ausgabe werden in Kapitel 9 genau beschrieben.

```
logical function sichern( datei, wert )
implicit none
character( len = 20 ), intent( in ) :: datei
real, intent( in )                  :: wert
integer                             :: iostat

    ! Der Rueckgabewert ist standardmaessig .false.
    sichern = .false.

    ! Datei oeffnen
    open( unit = 10, file = datei, iostat = iostat )

    ! Wenn das Oeffnen der Datei fehlgeschlagen ist,
    ! Funktion beenden (Rueckgabewert ist immer noch .false.).
    if( iostat /= 0 ) return

    ! Oeffnen war erfolgreich => in die Datei schreiben
    write( unit = 10, fmt = * ) wert

    ! Datei schliessen
    close( unit = 10 )

    ! alles hat geklappt => Rueckgabewert auf .true.
    sichern = .true.

end function sichern
```

6.5.2 Rekursive Unterprogramme

Es gibt Fälle, in denen es Sinn macht, dass sich ein Unterprogramm selber aufruft, dies ist z.B. bei einigen Such- und Sortieralgorithmen der Fall. Ein solches Unterprogramm nennt man **rekursiv**. Fortran erlaubt rekursive Unterprogramme jedoch nicht standardmäßig. Um sie dennoch zu ermöglichen, gibt es das Schlüsselwort `recursive`, dass einfach der `function`- oder `subroutine`-Anweisung des rekursiven Unterprogramms vorangestellt wird.

Beispiel

Hier ein triviales Beispiel für eine „rekursive Zählschleife“

```
recursive subroutine schleife( zaehler )
implicit none
integer, intent( in ) :: zaehler

    ! Rekursiver Aufruf (es muss auch eine Abbruchbedingung geben,
    ! da sonst die Rekursion solange fortgefuehrt wird, bis der
    ! Speicher voll ist)
    if( zaehler > 0 ) call schleife( zaehler-1 )

end subroutine schleife
```

Achtung! Im Zusammenhang mit der `print`-Anweisung, kann man leicht ungewollt einen rekursiven Aufruf erzeugen:

```
print *, machWas()
```

Wenn `machWas()` selbst wieder `print` aufruft, handelt es sich um einen rekursiven Aufruf von `print`, den Fortran nicht erlaubt.

Obwohl rekursive Unterprogramme bei manchen Problemen stark zu deren Vereinfachung beitragen können, bergen sie auch einige Gefahren. So kann die Logik von rekursiven Unterprogrammen oftmals recht schwer verständlich sein. Außerdem sind Programme mit rekursiven Aufrufen sehr speicherhungrig, da bei jedem Aufruf des rekursiven Unterprogramms ein neuer Satz von lokalen Variablen im Speicher angelegt werden muss. Im obigen Beispiel ist das nur die Variable `zaehler`, die vier Byte belegt. Soll jetzt die Schleife eine Million mal durchlaufen werden, ergibt das einen Speicherbedarf von vier Megabyte!

6.5.3 Das save-Attribut

Normalerweise „vergessen“ Unterprogramme zwischen zwei Aufrufen die Werte ihrer Variablen. Mit Vergessen ist gemeint, dass der Speicherplatz, der bei einem Unterprogrammaufruf für die lokalen Variablen des Unterprogramms reserviert wird, am Ende des Unterprogramms wieder freigegeben wird. Bei einem neuen Unterprogrammaufruf bekommen die Variablen dann u.U. einen ganz neuen Platz im Speicher(können somit insbesondere nicht auf alte Werte zurückgreifen). Dieses Vergessen kann man aber mit dem `save`-Attribut verhindern. Dann wird der Speicherplatz der lokalen Variablen, die das `save`-Attribut haben, nach Ende des Unterprogramms nicht gleich wieder freigegeben, sondern geschützt, so dass das Unterprogramm auch beim nächsten Aufruf auf vorherige Werte zugreifen kann.

Dabei ist es oft zweckmäßig, jede Variable, die das `save`-Attribut verwendet, mit einem Startwert zu versehen. Dieser Startwert wird der Variablen mit dem save-Attribut beim *allerersten* Unterprogrammaufruf zugewiesen. Bei allen späteren Aufrufen hat dieser Startwert *keine* Bedeutung mehr. Die Variable mit dem `save`-Attribut wird dann immer genau den Wert haben, den es am Ende des jeweils vorigen Unterprogrammdurchlaufes hatte. Man kann den Startwert auch weglassen, aber dann muss man dafür Sorge tragen, dass man der Variablen mit dem `save`-Attribut beim ersten Aufruf einen sinnvollen Wert zuweist, bevor man die Variable liest! Das wird in einem späteren Beispiel (Beispiel 6.5.4) gezeigt.

Dabei ist zu beachten, dass *nur die lokalen Variablen* einer subroutine bzw. einer function das `save`-Attribut erhalten können, nicht aber die Formalparameter, also die Variablen aus der Übergabezeile.

Dies würde nämlich sowohl zu inhaltlichen als auch zu technischen Konflikten führen.
Eine Variable mit dem `save`-Attribut soll bei einem Unterprogrammaufruf den Wert aus dem letzten Aufruf gespeichert haben, während ein Formalparameter mit dem `intent(in)`-Attribut bei einem Unterprogrammaufruf gerade über den Wert des zugehörigen Aktualparameters verfügen können soll. Aber auch bei den anderen Formalparametern ist es aus Gründen der technischen Realisierung der Übergabe nicht möglich, ihnen das save-Attribut zu geben.

Beispiel:

Dieses kurze Programm gibt die Zahlen von eins bis zehn aus.

```
program zaehle
implicit none

integer :: i

interface
    subroutine ausgabe
    implicit none
    end subroutine ausgabe
end interface

    do i=1,10
        call ausgabe
    end do

end program zaehle


subroutine ausgabe
implicit none

! Beim ersten Aufruf von ,ausgabe' hat i den Wert 1,
! die nachfolgenden Aufrufe koennen sich an den
! jeweils letzten Wert von i ,erinnern'.
integer, save :: i = 1

    print *,i
    i = i + 1

end subroutine ausgabe
```

Dieses Beispiel demonstriert schön einfach die Benutzung des `save`-Attributes. Den Sinn macht es leider noch nicht recht klar. Es ist mit den wenigen Mitteln, die uns bisher zur Verfügung stehen, aber auch noch schwierig, sinnvolle Beispiele zu finden.
Ein wichtiges Beispiel, in dem man auf das `save`-Attribut nicht verzichten kann, ist folgendes: Angenommen, es gibt ein Programm, das Lagerbestände oder aber Bestellungen von Artikeln verwaltet. Dann braucht dieses Programm eine Kostenfunktion, also eine Funktion, die die Kosten, der bestellten Artikel berechnet. Preise ändern sich aber ständig. Daher dürfen die Preise nicht fest in den Programmcode geschrieben werden, sondern müssen separat in einer Datei gespeichert werden, damit man sie bequem ändern kann.
Die Kostenfunktion muss dann aber, um Kosten berechnen zu können, erst einmal die Preislisten aus der Datei einlesen. Da Lesevorgänge sehr lange dauern und die Kostenfunktion bei einem Programmstart bestimmt sehr oft aufgerufen wird, wäre es sehr schlecht, d.h. zeitaufwendig, wenn die Kostenfunktion

jedes Mal wieder die Datei öffnen und die Preislisten auslesen müsste.

Also muss die Funktion so realisiert werden, dass sie nur beim ersten Aufruf die Preislisten einliest und sie sich dann *merkt*. Daher müssen die Variablen, die die Preisliste speichern, mit dem `save`-Attribut versehen werden. Dies geschieht in folgender Funktion, die zwar schon auf die noch nicht behandelten Dateioperationen zurückgreift, aber dennoch verständlich sein sollte.

6.5.4 Beispiel:

Diese kurze Funktion berechnet die Kosten für drei Stähle.

```
function kosten(stahl1,stahl2,stahl3)
    implicit none

        !   Variablen zur Verwaltung von Bestellmengen fuer 3 Staehle:
        real,intent(in)::stahl1,stahl2,stahl3
        real::kosten         !   Rueckgabewert

        !    Variablen zur Verwaltung von Preisen fuer 3 Staehle
        !    brauchen das save-Attribut, ohne Initialisierung:
        real,save::preis1,preis2,preis3

        !    merkt sich, ob die Preisliste eingelesen wurde;
        !    braucht das save-Attribut mit Initialisierung false,
        !    denn beim ersten Funktionsaufruf ist die Datei
        !    mit den Preisen noch NICHT gelesen
         logical,save::datGelesen=.false.

    !    wenn noch keine Preis-Daten gelesen wurden, einlesen:
    if(.not.datGelesen) then

        !    Datei oeffnen, Stahlpreise einlesen, Datei schliessen:
        open (unit=20,file='preise.dat')
        read(unit=20, fmt=*)preis1,preis2,preis3
        close(unit=20)

        datGelesen=.true.    !    ...Daten schon gelesen
    end if

    !    Kostenaufsummierung
    kosten=stahl1*preis1+stahl2*preis2+stahl3*preis3

end function kosten
```

Auf der PPM-Homepage findet sich noch eine ausgebaute Version dieser Funktion, in der neben dem `save`-Attribut noch selbstdefinierte Datentypen und Konstrukte zur Fehlerbehandlung bei Dateioperationen verwendet werden. Diese kann man sich nach Behandlung der entsprechenden Kapitel des Skriptes noch einmal genauer ansehen.

6.5.5 Unterprogramme als Parameter

Es ist auch möglich ganze Unterprogramme als Argumente an andere Unterprogramme zu übergeben. Auf diese Weise wird es möglich Algorithmen zu entwerfen, die nicht nur auf Datenmengen, sondern auch auf Funktionen- (bzw. Unterprogramm-) Mengen operieren. Ein klassisches Beispiel hierfür ist eine Funktion zur numerischen Bestimmung des bestimmten Integrals einer Kurve (oft mit Hilfe der Trapezregel):

```
program flaecheninhalt
implicit none

! Interfaces bei Unterprogrammen, die andere Unterprogramme als
! Parameter uebernehmen, tendieren dazu sehr lang zu werden, da
! sie geschachtelt werden muessen.
interface

    ! parabel ist die zu integrierende Funktion
    double precision function parabel( x )
    implicit none
    double precision, intent( in ) :: x
    end function parabel

    ! integral integriert beliebige Funktionen, die einen
    ! double precision parameter uebernehmen und den Rueckgabetyp
    ! double precision besitzen
    double precision function integral( f, a, b, n )
    implicit none
    double precision, intent( in )  :: a, b
    integer, intent( in )           :: n

    ! hier muss das Interface fuer die zu uebergebene Funktion
    ! definiert werden
    interface
        double precision function f( x )
        implicit none
        double precision, intent( in ) :: x
        end function f
    end interface

    end function integral

end interface

    print *, 'Die Flaeche unter der Parabel zwischen -2 und 2', &
            & ' (bei 100 Stuetzstellen) betraegt: ', &
            & integral( parabel, -2.0, 2.0, 100 )

end program flaecheninhalt
```

An dem Hauptprogramm mit seinem Interfaceblock erkennt man schon alle nötigen Elemente für die Verwendung von Unterprogrammparametern. Es werden hier zwei Funktionen definiert: `parabel`, die die zu integrierende Funktion darstellt und `integral`, die die Trapezregel implementiert. `integral` besitzt einen Funktionsparameter `f`, der in einem weiteren, zu `integral` gehörenden, Interfaceblock näher

beschrieben werden muss. Damit im späteren Programmverlauf `parabel` als Funktionsparameter an `integral` übergeben werden kann, müssen die Interfaces von `parabel`, außerhalb von `integral`, und von `f`, innerhalb von `integral`, bis auf den Namen exakt übereinstimmen.

Hier noch der Rest des Programmes:

```
double precision function parabel( x )
implicit none
double precision, intent( in ) :: x

    ! simpel
    parabel = x**2

end function parabel


double precision function integral( f, a, b, n )
implicit none

double precision, intent( in )  :: a, b
integer, intent( in )           :: n
double precision                :: h, summe
integer                         :: i

interface
    double precision function f( x )
    implicit none
    double precision :: x
    end function f
end interface

    ! Trapez-Algorithmus
    h = (b-a)/n
    summe = (f(a)+f(b))/2.0
    do i=1,n-1
        summe = summe + f( a+h*i )
    end do

    integral = h * summe

end function integral
```

6.6 Einige interne Fortran-Funktionen

Fortran95 bietet eine ganze Reihe fest eingebauter Funktionen, die dem Programmierer jederzeit zur Verfügung stehen und die Arbeit erheblich erleichtern können. Hier eine Auflistung einiger dieser Funktionen:

Ergebnis	Funktion	Parameter	Wirkung
ganz	`int`	(x) numerisch	liefert ganzzahligen Anteil von x durch Abschneiden
ganz	`nint`	(x) reell	liefert ganzzahligen Anteil von x durch Runden
reel	`real`	(x) reell	liefert reelle Zahlendarstellung für x
numerisch	`abs`	(x) numerisch	liefert Absolutwert(Betrag) von x
reell	`mod`	(x, y) reell	liefert Divisonsrest $x - \int(x/y) * y$
reell	`sin`	(x) reell	liefert $\sin(x)$ für x im Bogenmaß
reell	`cos`	(x) reell	liefert $\cos(x)$ für x im Bogenmaß
reell	`tan`	(x) reell	liefert $\tan(x)$ für x im Bogenmaß
reell	`asin`	(x) reell	liefert $\arcsin(x)$ im Bogen $-\pi/2$ bis $\pi/2$
reell	`acos`	(x) reell	liefert $\arccos(x)$ im Bogen $-\pi/2$ bis $\pi/2$
reell	`atan`	(x) reell	liefert $\arctan(x)$ im Bogen $-\pi/2$ bis $\pi/2$
reell	`exp`	(x) reell	Exponentialfunktion zur Basis e
reell	`log`	(x) reell	$\ln(x)$ Logarithmus zur Basis e, $x > 0$
reell	`log10`	(x) reell	$\log(x)$ Logarithmus zur Basis 10, $x > 0$

7 Felder

Felder (auch **Arrays** oder **indizierte Variable** genannt) dienen zur Verwendung mehrerer Speicherplätze unter einem Namen. Um die Grenzen der Möglichkeiten beim Gebrauch von **skalaren Variablen** zu verdeutlichen, werde zunächst das folgende Beispiel betrachtet.

Beispiel:

Ein Getränke-Vertrieb möchte den Jahres- und den durchschnittlichen Monatsumsatz von Bier ermitteln. Dazu sind die einzeln vorliegenden Monatsumsätze aufzusummieren und durch 12 zu dividieren.

Lösung mit skalaren Variablen: alle Monatsumsätze sind in einzelnen Variablen gespeichert; z.B. der Umsatz vom Juli in `umsatz_juli`:

```
implicit none

double precision :: umsatz_jan, umsatz_feb, umsatz_maerz, &
                  & umsatz_apr, umsatz_mai, umsatz_juni,  &
                  & umsatz_juli, umsatz_aug, umsatz_sep,  &
                  & umsatz_okt, umsatz_nov, umsatz_dez,   &
                  & jahres_umsatz, mittel_umsatz

! Monatsumsätze irgendwie einlesen:
 ⋮

! Berechnung des Jahresumsatzes:
jahres_umsatz = umsatz_jan + umsatz_feb + umsatz_maerz +   &
               & umsatz_apr + & umsatz_mai + umsatz_juni + &
& umsatz_juli + umsatz_aug + umsatz_sep +   &
& umsatz_okt + umsatz_nov + umsatz_dez

! Berechnung des durchschnittlichen Monatsumsatzes:
mittel_umsatz = jahres_umsatz / 12
```

In diesem Beispiel muss für jeden Monatsumsatz eine eigene Variable angelegt werden. Schon für nur 12 Monate ist das bereits recht lästige Schreibarbeit. Wenn z.B. der Tagesdurchschnittswert der Niederschläge eines Jahres berechnet werden soll, müssen sogar 365 skalare Variable eingetippt werden. Das ist nicht mehr zumutbar, und für derartige Probleme kommen Felder zum Einsatz.

7.1 Vereinbarung von Feldern, Abspeicherung

Die Datenstruktur „Feld“ wurde ursprünglich in die Programmiersprachen aufgenommen, um einen adäquaten und einfachen Umgang mit Matrizen und Vektoren zu ermöglichen. Mehrere Elemente gleichen Datentyps werden zusammengefasst und können unter einem gemeinsamen Namen angesprochen werden. Der Name eines Feldes muss den üblichen Konventionen für symbolische Namen (vgl. Abschnitt 4 Grundelemente der Programmiersprache Fortran95) genügen.

Ein Feld wird wie eine skalare Variable zu Beginn des Programmes (oder etwas präziser: vor der ersten ausführbaren Anweisung einer Programmeinheit) vereinbart. Dabei wird der Typ der Feldelemente und die „Größe“ des Feldes angegeben, damit der für ein Feld benötigte Speicherplatz reserviert werden

kann. Dies geschieht mittels des **`dimension`-Attributes**[17], das bei der Typ-Vereinbarung verwendet wird, oder alternativ durch Angabe der Größe in Klammern hinter dem Variablennamen bei der Typ-Vereinbarung.

Beispiele für Feldvereinbarungen:

```
double precision, dimension( 8 )     :: vektor
double precision, dimension( 8, 8 )  :: Matrix
double precision :: nochEinVektor( 8 ), nochEineMatrix( 8, 8 )
```

Durch die erste Anweisung werden acht Speicherplätze vom Typ `double precision` reserviert, die unter dem Namen `vektor` zugänglich sind. Die zweite Anweisung reserviert $8 \cdot 8 = 64$ Speicherplätze vom Typ `double precision`, die unter dem Namen `Matrix` erreichbar sind. Es ist naheliegend, sich darunter eine (8×8)-Matrix vorzustellen. Die dritte Anweisung ist äquivalent zur ersten bzw. zweiten Anweisung; auf diese Weise können Variablen unterschiedlicher Dimension in einer Zeile vereinbart werden.

An die **Feldelemente** „kommt man heran“, indem an den symbolischen Feldnamen in Klammern stehende Indizes angehängt werden: das erste Element des Feldes `vektor` ist `vektor(1)`, das zweite Element des Feldes `vektor` ist `vektor(2)` usw. bis zum letzten Element `vektor(8)`. Die derart selektierten Feldelemente werden auch indizierte Variable genannt (vgl. Abschnitt 7.2). Unter Verwendung eines Feldes kann das Getränke-Vertrieb-Beispiel einfacher realisiert werden.

Beispiel:

Lösung mit einem Feld (mit indizierten Variablen): alle Umsätze sind in einem Feld `monats_umsatz` mit 12 Elementen gespeichert. Auf die einzelnen Elemente wird durch einen Index `lauf` zugegriffen. Der Umsatz des Monats Juli (siebter Monat) ist z.B. gespeichert in `monats_umsatz(7)`:

```
implicit none

integer :: lauf
double precision, dimension(12) :: monats_umsatz
double precision :: mittel_umsatz, jahres_umsatz

! Umsätze irgendwie einlesen:
⋮

! jahres_umsatz initialisieren:
jahres_umsatz = 0.0

! alle Monatsumsätze aufsummieren mit Zählschleife:
do lauf = 1, 12
   jahres_umsatz = jahres_umsatz + monats_umsatz( lauf )
end do

mittel_umsatz = jahres_umsatz / 12
⋮
```

Diese Variante der Bierumsatzermittelung ist übersichtlicher und vor allem bei einer großen Anzahl von Variablen/Elementen kürzer.

[17] Es gibt noch weitere Möglichkeiten zur Vereinbarung eines Feldes, die in Fortran95 zulässig sind, die aber in diesem Umdruck nicht verwendet werden.

In der Numerik sind Felder der Länge 1000 oder mehr keine Seltenheit. Ohne die Verwendung von Feldern sind derartige Probleme nicht zu bewältigen.

Die allgemeine **Definition des `dimension`-Attributes** zur Feldvereinbarung lautet:

```
dimension(dim [, dim] ...)
```

dim legt die Grenzen einer Dimension und damit ihren Umfang fest und hat folgende Form:

```
[lower :] upper
```

lower spezifiziert den kleinsten (Index-)Wert (untere Grenze) einer Dimension und muss ein ganzzahliger Ausdruck größer, kleiner oder gleich Null sein. Falls dieser Wert nicht angegeben ist, wird er gleich 1 angenommen. *upper* spezifiziert den größten (Index-)Wert (obere Grenze) einer Dimension und muss ein ganzzahliger Ausdruck, größer, kleiner oder gleich Null, sein. Er muss außerdem größer oder gleich der unteren Grenze sein.

Damit erhält das `dimension`-Attribut die etwas ausführlichere Form:

```
dimension([lower:]upper[, [lower:]upper] ...)
```

lower und *upper* können zwar Ausdrücke sein, bei der Vereinbarung eines Feldes müssen diese aber auswertbar sein, um dem Compiler die exakte Größe des zu reservierenden Speicherplatzes mitzuteilen.

Falls die exakte Größe des Feldes erst zur Laufzeit, also insbesondere *nicht* zum Zeitpunkt der Kompilierung bekannt ist, muss zusätzlich das Attribut `allocatable` oder `pointer` verwendet werden, genaueres dazu wird in Abschnitt 7.4 Dynamische Speicherplatzverwaltung erläutert.

Der Umfang einer einzelnen Dimension ergibt sich aus:

```
umfang = upper - lower + 1
```

Die Maximalzahl der Dimensionen *dim*, die für ein Feld vereinbart werden darf, beträgt 7. Typischerweise werden aber nur eindimensionale Felder (für **Vektoren**) und zweidimensionale Felder (für **Matrizen**) benötigt.

Die Anzahl der Dimensionen wird im Fortran95-Standard als **Rang**[18] (engl. *rank*) bezeichnet. Die sogenannte **Gestalt** (engl. *shape*) eines Feldes wird bestimmt durch die Anzahl der Dimensionen und den Umfang jeder Dimension.

Ein paar Beispiele:

- die Anweisung

  ```
  integer, dimension(0:5) :: vec
  ```

  vereinbart ein eindimensionales Feld mit sechs `integer`-Elementen (der Umfang u des Feldes ist $5 - 0 + 1 = 6$); das erste Element des Feldes ist `vec(0)`, das letzte Element des Feldes ist `vec(5)`;
- die Anweisung

  ```
  integer, dimension( 4 , -1:2 ) :: Mat
  ```

[18] Eine etwas unglückliche Bezeichnung – der Rang eines Arrays hat natürlich nichts mit dem Rang einer Matrix im Sinne der linearen Algebra zu tun.

vereinbart ein zweidimensionales Feld; die erste Dimension hat einen Umfang von $4 - 1 + 1 = 4$, die zweite Dimension hat einen Umfang von $2 - (-1) + 1 = 4$; das Feld `mat` kann also angesehen werden als eine Matrix mit 4 Zeilen und 4 Spalten (und sehr ungewöhnlichen Elementindizes); die obige Anweisung reserviert also $4 * 4 = 16$ Elemente vom Typ `integer`.

Vektoren werden im allgemeinen als eindimensionale Felder, Matrizen als zweidimensionale Felder dargestellt.

- die Anweisung

```
real, dimension( 6 , 0:4 , 2:4 ) :: raum3
```

vereinbart ein dreidimensionales Feld mit 6 Zeilen, 5 Spalten und 3 Ebenen. Damit wird insgesamt Speicher für $6 * 5 * 3 = 90$ Elemente vom Typ `real` reserviert.

Ein Feld belegt also so viele Speicherplätze, wie sich aus dem Produkt der Umfänge aller Dimensionen ergibt.

Hinweis:

Die in den obigen Beispielen verwendeten Indexgrenzen sind in Fortran95 (im Gegensatz zu anderen Sprachen) möglich und erlaubt. Trotzdem muss vor solchen Spielereien dringendst gewarnt werden: Indexgrenzen sollten aus Gründen der Übersichtlichkeit immer von 1 bis n laufen und nicht von 0 bis $n - 1$ oder von -1 bis $n - 2$. Von dieser Regel sollte nur abgewichen werden, wenn

- es wirklich erforderlich ist (was nur äußerst selten der Fall sein dürfte) und
- man genau weiß, was man tut!

Die Vereinbarung eines Vektors bzw. einer Matrix sieht dann im Normalfall immer so aus:

```
double precision, dimension( 6 ) :: vektor_6
double precision, dimension( 6 , 6 ) :: Matrix_6
```

Dann können Formeln direkt aus einem Mathebuch abgetippt werden, und es ist erfahrungsgemäß mit den geringsten Problemen zu rechnen.

Bei einigen Anwendungen ist es für den Benutzer wichtig zu wissen, in welcher Reihenfolge die Elemente eines Feldes im Speicher des Rechners abgelegt sind. Grundsätzlich können die Elemente im Speicher aufgrund dessen physikalisch sequentieller Struktur nur linear hintereinander angeordnet sein. Für ein eindimensionales Feld ist die Reihenfolge trivial, sie entspricht der Reihenfolge der Indizierung. Bei einem zweidimensionalen Feld sind die Elemente aber ebenfalls sequentiell angeordnet und zwar in Fortran95 *spaltenweise*[19]. Ein Feld `matrix`, das folgendermaßen vereinbart wurde:

```
double precision, dimension(2,3) :: matrix
```

hat 2 Zeilen und 3 Spalten, d.h. $2 * 3 = 6$ Elemente, die man sich im allgemeinen so vorstellt:

$$\text{Matrix} = \begin{bmatrix} a(1,1) & a(1,2) & a(1,3) \\ a(2,1) & a(2,2) & a(2,3) \end{bmatrix}$$

[19] Das ist in C gerade umgekehrt, was gelegentlich als *Verbrechen* oder zumindest als *Sünde* angeprangert wird.

Abgespeichert werden die Elemente im Speicher in der folgenden Reihenfolge:

```
- a(1,1) - a(2,1) - a(1,2) - a(2,2) - a(1,3) - a(2,3) -
```

Allgemein gilt in Fortran95 bei mehrdimensionalen Feldern, dass bei der Speicherung der erste Index am schnellsten erhöht wird und der letzte Index am langsamsten.

In Abschnitt 7.6 Übergabe von Feldern an Prozeduren, variable Dimensionierung wird auf die Bedeutung dieser Anordnung der Feldelemente im Speicher näher eingegangen.

7.2 Indizierung der Variablen, Zugriff auf Feldelemente

Verwendet man Felder (indizierte Variable), so ist zu unterscheiden zwischen dem **Zugriff auf** ein gesamtes Feld durch den symbolischen Feldnamen und dem Zugriff auf einzelne Feldelemente durch den Feldnamen und die (in Klammern stehenden) Indizes.

Beispiele:

- Beispiel 1 – Hier wird auf das komplette Feld `raum3` zugegriffen:

```
double precision, dimension( 3 , 3 , 3 ) :: raum3
⋮
call sub( raum3 ) ! Übergabe des gesamten Feldes
⋮
```

- Beispiel 2 – Hier wird auf ein einzelnes Feldelement zugegriffen: dem Feldelement mit dem Index 3 wird der Wert 2.189 zugewiesen:

```
real, dimension( 500 ) :: vec
⋮
vec( 3 ) = 2.189
⋮
```

- Beispiel 3 – dem gesamten Feld (s.a. Abschnitt 7.7 Feldoperationen), also *jedem* Feldelement wird ein Wert zugewiesen:

```
double precision, dimension( 300 , 300 ) :: A
⋮
A = 0.0     ! alle Elemente von A mit 0.0 initialisieren
⋮
```

Die allgemeine Form des Zugriffs auf einzelne Feldelemente lautet:

array(*index* [, *index*] ...)

`array` ist ein symbolischer Feldname und *index* ein ganzzahliger Indexausdruck. Jeder Indexausdruck muss einen Wert haben, der innerhalb des Umfanges der entsprechenden Dimension liegt.

Ein wesentliches Merkmal der Datenstruktur Feld ist, dass der Index in einem Ausdruck *berechnet* werden kann. Verwendet wird diese Eigenschaft der indizierten Variablen gegenüber nicht-indizierten Variablen z.B. bei einer `do`-Anweisung.

Beispiel:

Dieses Beispiel berechnet die Euklidische Länge `length` eines Vektors mit 16 Komponenten, die in den ersten 16 Elementen des Feldes `vector` abgespeichert sind:

```
⋮
double precision, dimension( 20 ) :: vector          ! Feldlänge: 20
double precision :: length, sqrsum
integer :: number, lauf
⋮
number = 16
⋮
sqrsum = 0.0
do lauf = 1, number
   sqrsum = sqrsum + vector( lauf ) * vector( lauf )
end do
length = sqrt ( sqrsum )
⋮
```

Hinweis: die Funktion `sqrt( )` berechnet die Quadratwurzel ihres Argumentes (engl.: *square-root*).

Am obigen Beispiel ist noch eine weitere Kleinigkeit zu erkennen: das eindimensionale Feld `vector` hat die Länge 20, kann also 20 `double precision`-Werte aufnehmen. Im obigen Beispiel werden aber nur die ersten 16 Elemente benutzt, die letzten 4 Speicherplätze also „verschenkt“. Eine **Unterschreitung** der **Feldgrenzen** ist möglich (aber Speicherplatz verschwendend), eine **Überschreitung** der Feldgrenzen durch den aktuellen Index führt jedoch zu einem Fehler:

```
⋮
double precision, dimension( 12 ) :: trap  ! Feld trap: Länge 12
double precision :: hit, aaaargh
⋮
index = 15
⋮
hit = trap (index - 1)              ! lese Feldelement trap( 14 )

trap (index + 1 ) = aaaargh         ! schreibe in Feldelement trap( 16 )
⋮
```

Für das Feld `trap` sind 12 Speicherplätze reserviert, es wird aber versucht, (lesend) auf den 14. Speicherplatz zuzugreifen. Dort kann keine relevante (das Feld `trap` betreffende) Information stehen, die Variable `hit` wird also nicht den erwarteten Wert enthalten. Im günstigsten Fall bricht das Programm an dieser Stelle mit einer Fehlermeldung ab – darauf kann man sich aber *nicht* verlassen! Möglicherweise rechnet das Programm auch mit einem sinnlosen Wert für `hit` weiter.

In der letzten Zeile des obigen Beispiels wird versucht, dem 16. Element des Feldes `trap` den Wert `aaaargh` zuzuweisen. Da das 16. Element von `trap` nicht existiert, kann das nicht sinnvoll sein. Im günstigsten Fall bricht das Programm an dieser Stelle wieder mit einer Fehlermeldung ab, im ungünstigsten Fall wird aber eine andere Variable ohne Warnung *überschrieben*!

Wenn das Programm mit den o.g. falschen Werten rechnet, wird man sich evtl. wundern, dass an einigen Stellen merkwürdige Werte herauskommen[20] – wenn man das merkt!

Feldgrenzenüberprüfung:

Falls das Programm aufgrund des oben beschriebenen Fehlers abbricht, lautet die Fehlermeldung:

```
segmentation fault
```

oder:

```
bus error
```

Die beiden Fehler werden dadurch verursacht, dass die vereinbarten Feldgrenzen überschritten werden. Um das zu vermeiden, müsste das Programm vor jedem Zugriff auf ein Feldelement überprüfen, ob es innerhalb der vereinbarten Feldgrenzen liegt.

In Java ist diese Überprüfung (die die Programmausführung erheblich verlangsamt) standardmäßig eingebaut und lässt sich auch nicht abschalten. In C muss man sich selbst um derartige Dinge kümmern, ansonsten geht C davon aus, dass man weiß, was man tut.

In Fortran95 wurde ein Sprachfeature eingebaut, das die Vorteile beider Varianten nutzt: die Feldgrenzenüberprüfung zur Laufzeit lässt sich an- und abschalten. Wird ein Programm mit der Option „`-C`“ kompiliert, ist die Feldgrenzenüberprüfung aktiviert, der Kompilierungsbefehl lautet dann z.B.:

```
f95 -C -o testprogramm testprogramm.f90
```

Nach dem Auftreten eines `segmentation fault` oder während der Programmentwicklung wird die Feldgrenzenüberprüfung durch Kompilieren des Programmes mit der Option `-C` aktiviert. Wenn später dem Programm vertraut wird und die Ablaufgeschwindigkeit erhöht werden soll, wird ohne die Option `-C` kompiliert, z.B.:

```
f95 -o fertig testprogramm.f90
```

Das Programm `fertig` läuft dann ohne Feldgrenzenüberprüfung zur Laufzeit.

7.3 Ein- und Ausgabe von Feldern, Implizite do-Anweisung

Felder können wie nicht-indizierte (skalare) Variable in Ein-/Ausgabelisten auftreten. Dabei ist zu beachten, dass durch den indizierten symbolischen Feldnamen auf ein einzelnes Feldelement und durch den nicht indizierten symbolischen Feldnamen auf das gesamte Feld in der Reihenfolge der Abspeicherung (s. Kapitel 7.1 Vereinbarung von Feldern, Abspeicherung) zugegriffen wird. Beispiel:

```
⋮
real, dimension(20) :: werte      ! Vereinbarung d. Feldes werte
real :: alfa, beta                ! skalare Variable
⋮
read * , beta, werte(13), alfa
⋮
```

[20] Etwas deutlicher: das Programm erzeugt Datenmüll.

hier wird u.a. ein Wert in das 13. Element des Feldes `werte` eingelesen, während

```
print * , werte , werte(7)
```

alle Elemente des Feldes `werte` in der Reihenfolge der Abspeicherung, also spaltenweise, und dann noch einmal das 7. Element ausgibt – und zwar alle Elemente ohne Zeilenvorschub auf einer Zeile (sofern diese breit genug ist).

Will man mehrere Elemente eines Feldes **ausgeben**, so kann man, anstatt die Elemente einzeln indiziert in einer Ein-/Ausgabeliste aufzuführen, die sog. implizite do-*Anweisung* verwenden:

```
print *, (werte(i), i = 5,15 )
```

druckt das 5., 6., . . . , 14., 15. Element des Feldes `werte` aus.

Die allgemeine Form der impliziten do-Anweisung lautet:

```
( liste, i = start, ende[ ,schritt] )
```

Die Bezeichner *i*, *start*, *ende* und *schritt* haben dieselbe Bedeutung wie bei der do-Zählschleife (siehe Abschnitt 5.7 Die Wiederholung mit Zählvorschrift (do-Zählschleife)), und *liste* ist eine Ein-/ Ausgabeliste.

Der zweite Vorteil der impliziten do-Anweisung besteht darin, dass damit die Ein-/Ausgabe (Inhalt und Anzahl der Datensätze) flexibler gestaltet werden kann. Man vergleiche:

```
⋮
integer :: a( 20 )
⋮
! Variante 1, mit impliziter do-Schleife:

read * , k, ( a(i), i = 1 , k )

! Variante 2, mit normaler do-Schleife:

read * , k          ! erst Anzahl der Werte einlesen

do i = 1 , k        ! Zählschleife von 1 bis k
   read * , a(i)    ! nacheinander alle Elemente einlesen
end do
⋮
```

Folgender Unterschied der beiden Varianten ist zu beachten: im ersten Fall können die Elemente fortlaufend in einer (oder mehreren) Eingabezeile(n) stehen, im zweiten Fall wird für jedes Element ein neuer Datensatz (hier: eine neue Eingabezeile) benötigt.

Das gilt analog auch für die Ausgabe von Feldern. Die Zeile:

```
print * , a( 1 ) , ( a( i ) , i = 3,6 ) , a(9)
```

entspricht, wenn alle Elemente der impliziten do-Anweisung ausgeschrieben würden, der Zeile:

```
print * , a(1), a(3), a(4), a(5), a(6), a(9)
```

Die zweite Variante ist im übrigen für große Felder gar nicht mehr umsetzbar: für einen Vektor der Länge 1000 müssten 1000 Feldelemente explizit in der Ausgabeliste auftauchen!

Eine implizite do-Anweisung muss als Element einer Ein-/Ausgabeliste immer vollständig, d.h. mit umschließender Klammer angegeben werden.

Geschachtelte, implizite do-Anweisung:

Implizite do-Anweisungen können geschachtelt werden, d.h. die Ein-/Ausgabeliste einer impliziten do-Anweisung darf selbst wieder eine implizite do-Anweisung enthalten. Bei einer derartigen Schachtelung verändert sich die innerste (erste) Laufvariable am schnellsten, die äußerste (letzte) am langsamsten. Angewendet wird die Schachtelung z.B. bei der Ein-/Ausgabe von mehrdimensionalen Feldern. Die folgende `print`-Anweisung:

```
⋮
real, dimension( 6,5 ) :: Matrix
⋮
print * , (( Matrix(i,j), j=1,5 ), i= 1,6 )
⋮
```

druckt die Elemente des Feldes `matrix` zeilenweise, also entgegen der spaltenweisen Anordnung im Speicher aus. Eine z.B. zweifach geschachtelte do-Anweisung hat die allgemeine Form:

```
((liste, i1=start1, ende1[, schritt1]), i2=start2, ende2[, schritt2])
```

Die Bedeutungen der Elemente entsprechen denjenigen der ungeschachtelten impliziten do-Anweisung.

Bei dem letztgenannten Beispiel werden die Elemente des Feldes zeilenweise, und fortlaufend hintereinander ausgedruckt, was ein schlecht lesbares Druckbild ergibt. Will man die Elemente so ausdrucken, dass ein „matrixähnliches“ Bild entsteht, so kann man eine Kombination von do-Anweisung und impliziter do-Anweisung benutzen:

```
⋮
real, dimension( 6,5 ) :: Matrix
⋮
do i=1,6                                ! schaltet Zeile weiter
   print * , ( Matrix(i,j),   j=1,5 ) ! Spaltenindex j läuft
end do
⋮
```

Die implizite do-Anweisung dient zum Ausdruck jeweils einer „Zeile“ des Feldes `Matrix`, während die gewöhnliche do-Anweisung die Zeilen weiterschaltet, da für jede Ausführung der `print`-Anweisung ein neuer Datensatz (eine neue Ausgabezeile) begonnen wird. Voraussetzung für die Lesbarkeit ist natürlich, dass die Elemente einer Zeile der Matrix auf eine Zeile des Ausgabegerätes (`xterm`) passen.

7.4 Dynamische Speicherplatzverwaltung

In den obigen Beispielen wird davon ausgegangen, dass die Größe eines Feldes zur Zeit der Kompilierung bekannt ist. Das ist in der Praxis meist nicht der Fall, dort sind $(n \times n)$-Matrizen und dazu „passende“ $(n \times 1)$-Vektoren zu behandeln, und die Größe von n ist variabel.

Mit den bisherigen Methoden müsste jedes Mal, wenn ein neues Problem mit einer anderen Ordnungszahl n auftaucht, die Deklaration der verwendeten Felder angepasst und das Programm neu kompiliert werden.

Zum Teil wird das auch tatsächlich so gemacht, oder es wird ein recht uneleganter Trick[21] benutzt, um dieses Problem in den Griff zu bekommen. Besser ist die Verwendung einer Methode, die den benötigten Speicherplatz für die Felder dann beschafft, wenn bekannt ist, wie groß die Felder tatsächlich sind. Das ist meist erst dann der Fall, wenn das Programm bereits läuft.

Dieses Feature heißt dynamische Speicherplatzverwaltung: der Speicher wird dynamisch (nach Bedarf) reserviert (**alloziert**) und nicht statisch (fest/unflexibel) zum Zeitpunkt des Kompilierens. In C ist dies schon lange möglich (allerdings ist die Syntax unzumutbar), jetzt bietet auch Fortran die dynamische Speicherallokation.

7.4.1 Das `pointer`-Attribut

Die Nutzung dieses Features ist denkbar einfach. Bei der Vereinbarung der Felder wird zusätzlich das **`pointer`-Attribut** verwendet, und die Umfänge der Dimensionen im `dimension`-Attribut werden durch den **Platzhalter „`:`“** ersetzt:

```
double precision, pointer, dimension( : ) :: vektor
double precision, pointer, dimension( : , : ) :: Matrix
integer, pointer, dimension( : , : , : , : ) :: vier_dim_feld
```

Durch diese Vereinbarung ist dem Compiler die Anzahl der Dimensionen (max. 7 sind erlaubt) bekannt, der Umfang jeder Dimension wird „nachgeliefert“. Wenn während der Programmausführung die Umfänge ermittelt (berechnet oder eingelesen) wurden, wird mit einer **`allocate`-Anweisung** der benötigte Speicherplatz vom Betriebssystem angefordert:

```
⋮
read * , n                  ! Ordnungszahl einlesen
⋮
allocate( vektor( n ) )     ! Speicher d. gewünschten Größe allozieren
allocate( Matrix( n , n ) ) ! dito
⋮
```

Jetzt kann genauso mit den Feldern bzw. mit den Feldelementen gearbeitet werden, als wenn sie statisch vereinbart worden wären.

Wenn die Felder zu einem späteren Zeitpunkt des Programmablaufes nicht mehr gebraucht werden, kann der Speicher mit einer **`deallocate`-Anweisung** wieder freigegeben werden:

[21] Die Felder werden möglichst groß vereinbart, benutzt wird dann aber nur ein Bruchteil davon.

```
   ⋮
   deallocate( vektor )
   deallocate( Matrix )
   ⋮
```

Insbesondere bei länger laufenden Programmen sollte Speicher immer freigegeben werden, wenn er nicht mehr benötigt wird. Sonst wird das Programm u.U. immer größer und verbraucht unnötig Systemressourcen.

Achtung!!!
Für eine Variable mit dem `pointer`-Attribut kann und darf KEIN `intent`-Attribut mehr vergeben werden.
Wenn das `pointer`-Attribut verwendet wird, MUSS ein `interface` verwendet werden.
Nachdem das Feld mit dem `allocate`-Befehl alloziert wurde kann in jedem Programmteil dem dieses Feld übergeben wird, damit gearbeitet werden.

Beispiel:

```
program DynamischeFelder
  implicit none
  integer,pointer,dimension(:,:) :: DynFeld

  interface
      subroutine FeldAllozieren (DynFeld)
      implicit none
      integer,pointer,dimension(:,:) :: DynFeld
      integer:: zeilen,spalten
      end subroutine FeldAllozieren
  end interface

  print*,'Dieses Programm alloziert ein Feld in einer Subroutine.'

  call FeldAllozieren(DynFeld)

  DynFeld=0

  print*,DynFeld

  deallocate(DynFeld)

end program DynamischeFelder




subroutine FeldAllozieren (DynFeld)
  implicit none
  integer,pointer,dimension(:,:) :: DynFeld
  integer:: zeilen,spalten

  print*,'Bitte geben sie die Anzahl an Zeilen an:'
  read*, zeilen
```

```
  print*,'Bitte geben sie die Anzahl an Spalten an:'
  read*, spalten

  allocate (DynFeld(zeilen,spalten))

end subroutine FeldAllozieren
```

Auch hier ist die Freigabe des reservierten Speichers am Ende des Programms mit `deallocate` erforderlich.

Es gibt jedoch 2 Möglichkeiten das `pointer`-Attribut zu verwenden.
Man kann ein Feld mit dem `pointer`-Attribut wie im obigen Beispiel gezeigt deklarieren:

```
integer,pointer,dimension(:,:)::Feld
```

oder man verwendet folgende Variante der Deklaration:

```
integer,pointer::Feld(:,:)
```

Es wäre jedoch zu empfehlen, die erste Variante zu benutzen, damit der syntaktische Aufbau im Einklang mit der normalen Felddeklaration steht.

7.4.2 Das `allocatable`-Attribut

Anstelle des `pointer`-Attributs kann man auch das `allocatable`-Attribut verwenden.
Der große Nachteil von `allocatable`-Feldern ist jedoch, dass das Feld in dem Programmteil alloziert werden muss, in dem es deklariert wurde.
Bei Fällen, in denen man ein Feld als dynamisch deklarieren und das Allozieren einem Unterprogramm überlassen möchte, sollte daher stets das `pointer`-Attribut verwendet werden.
Das `allocatable`-Attribut legt genauso wie das `pointer`-Attribut ein dynamisches Feld an, dessen Größe noch nicht bekannt ist.
Bis auf den Unterschied im Deklarationsteil:

```
  integer,allocatable,dimension(:,:) :: DynFeld
```

statt

```
  integer,pointer,dimension(:,:) :: DynFeld
```

gilt in beiden Fällen der gleiche Satz an Befehlen, siehe vorigen Abschnitt: `allocate` und `deallocate`.

7.5 Die `size`-Funktion

Im Zusammenhang mit dynamischer Speicherverwaltung und vor allem mit der Übergabe von Feldern an Unterprogramme (siehe Kapitel 7.6) ist die `size`-Funktion von Fortran sehr wichtig. Mit ihr lässt sich die Dimension eines Feldes (dynamisch oder statisch) zur Laufzeit abfragen:

```
    size( feld [, dim] )
```

`feld` ist dabei ein beliebigdimensionales statisches oder dynamisches Feld. Ohne das optionale Argument `dim` liefert `size` die Gesamtzahl der Feldelemente zurück. Ist `dim` angegeben, wird nur die Größe der entsprechenden Dimension zurückgegeben.

Beispiel:

```
program feldgrenzen
implicit none
integer, dimension( 10, 20 )    :: feld

    print *, size( feld )       ! => 200
    print *, size( feld, 1 )    ! => 10
    print *, size( feld, 2 )    ! => 20

end program feldgrenzen
```

7.6 Übergabe von Feldern an Prozeduren, variable Dimensionierung

Die symbolischen Namen von Feldern können wie skalare (nicht-indizierte) Variable in der Parameterliste von Prozeduren (Funktionen und Subroutinen) auftreten. Übergeben wird nur der Feldname ohne Indizes:

```
⋮
real, dimension( 22 ) :: feld
⋮
call sub(feld)               ! Übergabe v. feld der Länge 22
⋮
```

Innerhalb der Prozedur muss die Größe des übergebenen Feldes ebenfalls vereinbart werden:

```
subroutine sub(array)
implicit none
real, dimension( 22 ) :: array ! wieder Länge 22 vereinbaren
                               ! Achtung: unpraktisch
⋮
```

Die Dimensionierung des Feldes in der Subroutine muss exakt derjenigen in der aufrufenden Programmeinheit (hier: dem Hauptprogramm) entsprechen, da sonst auf falsche Speicherplätze zugegriffen werden kann!

Ein warnendes Beispiel:

Falls einem Unterprogramm die falschen Feldgrenzen übergeben werden, entsteht bei mehrdimensionalen Feldern eine schwerwiegende Fehlermöglichkeit, die im folgenden Beispiel erläutert wird:

```
⋮
! Vereinbarung der Variablen im Hauptprogramm
implicit none
integer, parameter :: dim_1 = 5 ! Zeilenanzahl
integer, parameter :: dim_2 = 5 ! Spaltenanzahl
integer :: m , n

! Mat ist eine (5x5)- bzw. (dim_1 x dim_2)-Matrix:
real, dimension( dim_1 , dim_2 ) :: Mat
⋮
read*, Mat (2,2)           ! weise Element (2,2) einen Wert zu
⋮
m=4 ; n=3

call sub ( Mat , m , n )   ! Aktualparameter:( Mat, 4 , 3 )
⋮
subroutine sub (Feld , g1 , g2 )
implicit none
integer :: g1 , g2         ! g1 = 4 , g2 = 3
real, dimension ( g1 , g2 ) :: Feld

! Die subroutine sub "denkt" jetzt, Feld sei eine (4x3)-Matrix

print * , Feld(2,2)
⋮
```

Jetzt sind `Mat(2,2)` im aufrufenden Programm und `Feld(2,2)` im Unterprogramm *nicht* identisch!

Genau genommen wird in der Parameterliste eines Unterprogramms nämlich nicht das ganze Feld, sondern die Startadresse des Feldes im Speicher übergeben. Ab dieser Startadresse holt sich das Unterprogramm alle weiteren Feldelemente von der Stelle, an der sie aufgrund der Dimensionierung (und des Datentyps) stehen müssten. Wenn einem Unterprogramm nun eine falsche Dimensionierung mitgeteilt wird, holt es sich ein bestimmtes Feldelement an der falschen Speicherstelle.

Im obigen Beispiel passiert folgendes: im Hauptprogramm sind für das Feld `Mat` $5 \cdot 5 = 25$ Speicherplätze reserviert, die spaltenweise in der Reihenfolge

1.	2.	3.	4.	5.	6.	7.	...
`Mat(1,1)`	`Mat(2,1)`	`Mat(3,1)`	`Mat(4,1)`	`Mat(5,1)`	`Mat(1,2)`	`Mat(2,2)`	...

abgespeichert sind. In der Subroutine hingegen wird wegen der variablen Dimensionierung mit `m = 4` und `n = 3` „angenommen“, dass das Feld `Feld` nur $4 \cdot 3 = 12$ Speicherplätze belegt. Die Reihenfolge der Elemente der (4×3)-Matrix wäre dann im Unterprogramm `sub` (spaltenweise):

1.	2.	3.	4.	5.	6.	...
`Feld(1,1)`	`Feld(2,1)`	`Feld(3,1)`	`Feld(4,1)`	`Feld(1,2)`	`Feld(2,2)`	...

Im Hauptprogramm belegt `Mat(2,2)` den 7. Speicherplatz, in der Subroutine `sub` belegt `Feld(2,2)` aber den 6. Speicherplatz, `Mat(2,2)` ist also nicht gleich `Feld(2,2)`.

Das Unterprogramm verwendet den falschen Wert!

Als Konsequenz daraus sollte bei der Dimensionierung von Feldern in Unterprogrammen höchste Sorgfalt aufgewandt werden. Auch die Feldgrenzenüberprüfung zur Laufzeit kann den obigen Fehler nicht aufdecken, da durch die falsche Dimensionierung im Unterprogramm keine Feldgrenzen überschritten werden.

Wenn Felder in Prozeduren exakt so dimensioniert werden müssen, wie in der aufrufenden Programmeinheit, führt das dazu, dass bei jeder Veränderung der Feldgröße *alle* entsprechenden Dimensionierungen in *jeder* Subroutine geändert werden müssen. Das ist nicht nur unbequem, sondern auch höchst fehlerträchtig. Jedoch gibt es Abhilfe: im Gegensatz zum Hauptprogramm müssen in Unterprogrammen die Feldgrenzen nicht zur Kompilierzeit feststehen, statt dessen kann wie bei dynamischer Speicherplatzverwaltung ein Platzhalter angeben werden:

```
⋮
subroutine sub( array )
implicit none

real, dimension( : ) :: array
⋮
```

Die Größe des übergebenen Arrays lässt sich dann mit Hilfe der `size`-Funktion bestimmen.

Zudem ist es in Unterprogrammen auch möglich Arrays mit variabler Größe ohne das `allocatable`-Attribut zu definieren. Das ist dann sehr sinnvoll, wenn man mit Funktionen arbeitet, die Arrays als Argumente und Rückgabewerte besitzen:

```
⋮
function func( array )
implicit none

real, dimension( : )                :: array
real, dimension( size( array ) )    :: func
⋮
```

Diese Funktion übernimmt ein Array vom Typ `real` beliebiger Größe und liefert ein anderes Array mit der gleichen Größe an die aufrufende Programmeinheit zurück.

Achtung! Obiges Beispiel lässt nur mit dem IBM-Compiler übersetzen, nicht mit dem Linux-Compiler – dieser kann keine Funktionen verarbeiten, die Arrays zurückgeben. Statt dessen muss auf die Rückgabe des Arrays über die Parameterliste zurückgegriffen werden (was i.A. bedeutet eine Subroutine zu verwenden).

7.7 Feldoperationen

In Fortran95 können viele Operationen und Funktionen, die normalerweise nur für skalare Variable definiert sind, auch auf Felder angewandt werden. Die Operationen wirken dann auf jedes Feldelement, was z.B. beim Initialisieren und Addieren von Matrizen einiges an Schreibarbeit erspart:

```
double precision, dimension( 30 , 30 ) :: A , B , C
integer :: i , j
⋮
! C herkömmlich initialisieren:
do i = 1 , 30
   do j = 1 , 30
      C(i,j) = 0.0
   end do
end do

! herkömmliche Matrizenaddition:
do i = 1 , 30
   do j = 1 , 30
      C(i,j) = A(i,j) + B(i,j)
   end do
end do

! Fortran95-Initialisierung und Matrizenaddition:
C = 0.0
C = C + B
⋮
```

Selbstverständlich müssen Felder, die addiert oder subtrahiert werden, die gleiche Gestalt haben, sonst sind die Operationen (wie bei einer per-Hand-Rechnung auch) sinnlos.

sum, product:

Die eingebauten Funktionen `sum(A)` und `product(A)` liefern die Summe und das Produkt aller Feldelemente des als Parameter übergebenen Feldes `A`:

```
⋮
double precision, dimension( 5 , 5 ) :: A
⋮
print * , ' Produkt aller Feldelemente von A =' , product(A)
print * , ' Summe aller Feldelemente von A =' , sum(A)
⋮
```

Die Summe eines Vektors kann z.B. zur Normierung desselben verwendet werden. Das folgende Beispiel berechnet die Summe der Elemente der ersten Spalte von `A`:

```
⋮
double precision, dimension( 5 , 5 ) :: A
⋮
print * , ' Summe aller Elemente der 1. Spalte von A =' , &
          & sum( A(:,1) )
⋮
```

maxval, minval:

Manchmal ist es von Bedeutung, das größte oder kleinste Element eines Feldes herauszufinden. In Fortran95 braucht man dieses Element nicht mit einem selbstprogrammierten Algorithmus zu suchen, sondern kann die vordefinierten Funktionen `maxval` und `minval` verwenden:

```
⋮
print * , ' größtes Element von A: ' , maxval( A )
print * , ' kleinstes Element von A:' , minval( A )
⋮
```

maxloc, minloc:

Auch der Ort bzw. die Feldindizes des größten oder kleinsten Feldelementes lassen sich einfach mit einer der beiden vordefinierten Funktionen `maxloc` oder `minloc` abfragen. Die Funktionen liefern einen Vektor zurück, der so viele Einträge wie das untersuchte Feld Dimensionen hat. Z.B. hat eine Matrix zwei Dimensionen, der von `maxloc` oder `minloc` zurückgelieferte Vektor hat dann zwei Elemente, die den Zeilen- und Spaltenindex des größten bzw. kleinsten Matrixelementes angeben:

```
⋮
double precision, dimension( 5 , 5 ) :: A
integer, dimension( 2 ) :: indexvek
⋮
indexvek = maxloc( A )
print * , ' das größte Element von A hat Indizes:' , indexvek

indexvek = minloc( A )
print * , ' das kleinste Element von A hat Indizes:' , indexvek
⋮
```

transpose:

Die eingebaute Funktion `transpose` dient dazu, eine Matrix zu transponieren. Dabei ist bei nicht quadratischen Matrizen darauf zu achten, dass die Gestalt der transponierten Matrix zur nichttransponierten Matrix „passt“, z.B. ist die Transponierte einer

```
⋮
double precision, dimension( 3 , 4 ) :: C
double precision, dimension( 4 , 3 ) :: D
⋮
D = transpose( C )
```

matmul:

Der Multiplikationsoperator `*` bewirkt auf Matrizen angewendet nicht etwa eine Matrizenmultiplikation, sondern es werden alle einander entsprechenden Matrixelemente skalar miteinander multipliziert (ähnlich der Addition von Matrizen):

```
⋮
double precision, dimension( 4 , 4 ) :: C , D , E
integer :: i , j
⋮
! Das braucht man nur selten:
do i = 1 , 4
   do j = 1 , 4
      E(i,j) = C(i,j) * D(i,j)
   end do
end do

! eine kürzere Schreibweise ist:

E = C * D
```

Erfreulicherweise ist in Fortran95 auch eine „richtige“ Matrizenmultiplikation vordefiniert, die Funktion `matmul`. Sie erhält zwei Felder als Argumente und liefert als Ergebnis das Matrizenprodukt der beiden zurück. Dabei ist darauf zu achten, dass die beiden als Parameter übergebenen Matrizen auch *verkettbar* sind (d.h. Spaltenzahl der ersten Matrix gleich Zeilenzahl der zweiten Matrix).

```
double precision, dimension( 4 , 3 ) :: B
double precision, dimension( 3 , 8 ) :: C
double precision, dimension( 4 , 8 ) :: A
⋮
A = matmul( B , C )
```

Der Sonderfall Matrix mal Vektor ist ebenfalls möglich und des öfteren bequem einzusetzen, etwa um schnell zu kontrollieren, ob der selbstentwickelte Löser für das Gleichungssystem $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ auch das richtige Ergebnis (oder besser: ein brauchbares[22] Ergebnis) liefert:

```
integer, parameter :: n = 33                ! Ordnungszahl
double precision :: epsilon = 1.E-9         ! Fehlertoleranz
double precision, dimension( n , n ) :: A   ! Koeffizientenmatrix
double precision, dimension( n ) :: x       ! Unbekanntenvektor
double precision, dimension( n ) :: b       ! rechte Seite
double precision, dimension( n ) :: res     ! Residuum
⋮
! A , b einlesen/aufstellen und x ermitteln:
⋮
! Probe Ax = b bzw. Residuum res = Ax - b klein ?
res = matmul( A , x ) - b

if( sum(res) < epsilon ) print * , ’ sieht gut aus ... :o)’
```

dot_product:

Auch das Skalarprodukt zweier Vektoren ist mit der Funktion `dot_product` vordefiniert. Die beiden als Parameter übergebenen Vektoren müssen selbstverständlich die gleiche Anzahl an Feldelementen haben. Im folgenden Beispiel wird die Länge eines Vektors mit vordefinierten Funktionen berechnet:

[22] Der Rechner liefert das exakte Ergebnis aufgrund der beschränkten Menge der Maschinenzahlen i.d.R. nicht.

```
⋮
integer, parameter :: n = 100
double precision, dimension( n ) :: vektor
double precision :: laenge
⋮
laenge = sqrt( dot_product( vektor , vektor ) )
```

Die hier exemplarisch vorgestellten Feldfunktionen sind häufig ganz brauchbar und ersparen den einen oder anderen Nachmittag an Programmieraufwand.

Einige der Funktionen haben noch weitere, optionale Parameter, die hier nicht erwähnt wurden. Für einen vollständigen Überblick über die Fortran95-Feldfunktionen muss auf ein Referenzhandbuch verwiesen werden.

8 Ein- und Ausgabemöglichkeiten

Die bis jetzt vorgestellten Möglichkeiten der Ein- und Ausgabe von Text bzw. von Zahlen sind aus folgenden Gründen bei vielen Anwendungen nicht ganz zufriedenstellend:

- Die Ein- und Ausgaben werden immer über den Bildschirm abgewickelt. Oft muss man deshalb gerade bei der Programmentwicklung, z.B. für Testläufe, viele Daten immer wieder von Hand eingeben. Es wäre eine große Erleichterung, wenn man den gesamten Datensatz nur einmal schreiben müsste, z.B. auf eine „Datendatei“ und beim Programmlauf von dieser Datei „lesen“ könnte.
- Oft möchte man die Ergebnisse eines Rechenlaufs als Eingabedaten für ein Folgeprogramm verwenden. Dazu müsste das Programm Ergebnisse auf eine Datei schreiben können, welche vom Folgeprogramm gelesen werden kann.
- Die Darstellung von `real`-Zahlen in der Exponentialdarstellung ist nicht so gut zu lesen und besonders für einen Laien schlecht zu interpretieren. Bei der Ausgabe von `integer`-Zahlen wird ein ziemlich großes Feld (10 Stellen) in der Ausgabe reserviert, das unter Umständen stört. Hier wünscht man sich eine größere Kontrolle über die Art der Darstellung.

8.1 Formatsteuerung bei der Ausgabe

Betrachten wir zunächst den letzten Punkt: Die Verbesserung der Kontrolle über die Ausgabe von Texten und Zahlen, um z.B. gut lesbare Tabellen zu erstellen. Der bisher ausschließlich verwendete Befehl zur Ausgabe von Ergebnissen war:

```
print *, v [, v] ...
```

wobei *v* sein konnte:

- eine `real`-, `double precision` oder `integer`-Variable
- eine `real`-, `double precision` oder `integer`-Konstante
- eine `character`-Variable
- eine `character`-Konstante oder ein `string`

Der Stern gibt an, dass für die Darstellung der Elemente der Ausgabeliste ein maschinenabhängiges Format benutzt wird. Um mehr Kontrolle zu haben, müssen eigene Formatbeschreiber verwendet werden. Das geschieht durch die **`format`-Anweisung**. (Man kann das auch anders als durch die `format`-Anweisung machen, was aber komplizierter und unübersichtlicher ist).

Die `format`-Anweisung ist eine nicht ausführbare Anweisung. Sie enthält Formatbeschreiber, die für jedes Element der Ausgabeliste vorschreiben, wie die Darstellung erfolgen soll. Die `format`-Anweisung darf an beliebiger Stelle in einer Programmeinheit stehen und wird durch eine Anweisungsnummer einer `print`-Anweisung zugeordnet.

Damit die einer `print`-Anweisung zugeordnete `format`-Anweisung auch bei längeren Programmen vom menschlichen Leser gefunden werden kann, ordnet man `format`-Anweisungen entweder direkt hinter der zugehörigen `print`-Anweisung oder aber am Ende der Programmeinheit (also der `subroutine` oder der `function`) hinter der `return`-Anweisung an, aber vor der `end`-Anweisung. Die zweite Möglichkeit wird dann verwendet, wenn eine `format`-Anweisung von mehreren `print`-Anweisungen verwendet wird.

Beispiel:

```
print * , x, y
```

wird ersetzt durch

```
print 10, x, y
```

10 ist die **Anweisungsnummer** oder „**Label**“ der format-Anweisung, in der die Ausgabe der beiden Variablen x und y vorgeschrieben wird:

```
10 format (liste der formatbeschreiber)
```

Formatbeschreiber werden unterschieden nach dem Typ der Variablen oder Konstanten in der Ausgabeliste, deren Ausgabe festgelegt werden soll:

- integer-Formatbeschreiber
- real-Formatbeschreiber
- double precision-Formatbeschreiber
- character-Formatbeschreiber
- complex-Formatbeschreiber

Daneben gibt es noch eine Reihe von Formatbeschreibern, die nicht einem Element der Ausgabeliste zugeordnet sind. Sie dienen zur Positionierung, z.B. bei der Ausgabe von Tabellen.

Der I-Formatbeschreiber:

Mit dem I-Formatbeschreiber wird die Ausgabe von integer-Zahlen in einer Ausgabeliste festgelegt. Er hat die Form:

```
Iw
```

wobei *w* eine natürliche Zahl ist, welche die Anzahl der Stellen angibt, die in der Ausgabe für die Darstellung der Zahl vorgesehen werden. Beispiel:

```
integer :: zahl1
zahl1 = 123
print 10, zahl1
10 format (I5)
```

erzeugt die Ausgabe:

```
␣␣123
```

Erklärung: Die Variable zahl1 hat den Integerwert 123. Sie wird in der Liste eines print-Befehls aufgeführt, bei der die Art der Darstellung durch die format-Anweisung mit dem Label 10 festgelegt wird. Durch die Formatbeschreibung I5 werden für die Darstellung von zahl1 5 Stellen vorgesehen, die Ausgabe erfolgt rechtsbündig und führende Nullen werden als Leerstellen ausgegeben.

Beispiel:

```
integer :: zahl1, zahl2
zahl1 = 123
zahl2 = 45
print 15, zahl1, zahl2
15 format (I5, I7)
```

erzeugt die Ausgabe:

```
␣␣123␣␣␣␣45
```

Erklärung: Da im `print`-Befehl zwei Elemente aufgeführt werden, müssen in der zugehörigen Formatanweisung auch zwei Formatbeschreiber angegeben werden. Die erste Variable `zahl1` wird im Format `I5`, die zweite im Format `I7` ausgegeben. Im Ausdruck erscheint `zahl2` rechtsbündig in einem 7 Druckstellen breiten Feld von der 6. bis zur 12. Spalte.

Der F-Formatbeschreiber:

Den F-Formatbeschreiber verwendet man, um `real`-Zahlen in der gewohnten Weise mit Vor- und **Nachkommastellen**, aber ohne Exponenten auszugeben. Er hat die Form:

F*w*.*d*

wobei *w* die Gesamtanzahl der Stellen angibt, die in der Ausgabe für die Darstellung einer `real`-Zahl vorgesehen werden. Der Dezimalpunkt zählt hierbei mit. *d* gibt an, wieviel Nachkommastellen gedruckt werden sollen. Beispiel:

```
real :: xwert
xwert = 10.5336
print 20, xwert
20 format (F8.2)
```

erzeugt die Ausgabe:

```
␣␣␣10.53
```

Erklärung: Für die Ausgabe der Variablen `xwert` mit dem Wert `10.5336` werden insgesamt 8 Stellen reserviert, wovon zwei (die 7. und 8. Position) Nachkommastellen sind. Der Dezimalpunkt steht also in der 6. Position. Der F-Formatbeschreiber wird sehr oft verwendet!

Der E-Formatbeschreiber:

Unter Umständen, wenn nämlich die Zahlenwerte sehr groß oder sehr klein sind, ist die Darstellung in **Exponentialschreibweise** der üblichen vorzuziehen. Dazu verwendet man den E-Formatbeschreiber:

E*w*.*d*

w hat die gleiche Bedeutung, wie beim F- bzw. `I`-Formatbeschreiber. *d* legt die Anzahl der **signifikanten Stellen** fest.

Beispiel:

```
real :: xwert
xwert = 9.85
print 25, xwert
25 format (E10.4)
```

erzeugt die Ausgabe:

```
␣.9850E+01.
```

Erklärung: Für die Ausgabe sind insgesamt 10 Stellen vorgesehen. Die letzten 4 Stellen sind immer für die Angabe des Buchstaben E zur Kennzeichnung der Exponentialdarstellung und für das Vorzeichen des Exponenten reserviert, sowie 2 Stellen für den Exponenten selbst. Der Exponent wird so gewählt, dass der Betrag der ausgegebenen Zahl in Gleitkommadarstellung immer < 1 und ≥ 0.1 ist. Da neben den vier signifikanten Stellen (im Beispiel) auch noch der Dezimalpunkt und eventuell ein Vorzeichen gedruckt werden kann, muss bei der Verwendung des E-Formatbeschreibers beachtet werden, dass $w \geq d + 6$ ist. Im Beispiel darf also die gesamte Feldweite für die Darstellung nicht kleiner als 10 gewählt werden!

Der A-Formatbeschreiber:

Die Ausgabe von `character`-Konstanten oder -Variablen wird durch den A-Formatbeschreiber definiert. Er hat die Form:

```
A[w]
```

Wenn *w* nicht angegeben wird, werden bei der Ausgabe genau so viele Stellen ausgegeben, wie die Charactervariable lang ist. Dazu das erste Beispiel:

```
character (len = 10) :: text1
character (len = 10) :: text2
character (len = 1) :: strich

text1='Fortran95'
text2='Kurs'
strich='-'

print 10, text1, strich, text2
10 format (A,A,A)
```

erzeugt die Ausgabe:

```
Fortran95␣-Kurs␣␣␣␣␣␣
```

Erklärung: Nach der Vereinbarung der Charactervariablen `text1`, `text2` und `strich` werden für die Ausgabe 10 Stellen für `text1` und `text2` und 1 Stelle für `strich` vorgesehen. Zu beachten ist, dass `text2` 10 Zeichen enthält: Die ersten vier bilden das Wort `Kurs`, die nächsten sechs die Leerzeichen und werden auch als solche ausgegeben!

Das zweite Beispiel zeigt, was man bei der Verwendung der zweiten Form des A-Formatbeschreibers mit Angabe der Feldweite zu beachten hat:

```
character (len = 20) :: text1
character (len = 16) :: text2
character (len = 12) :: text3

text1 = 'Projektgruppe'
text2 = 'Praktische'
text3 = 'Mathematik'

print 10,text1
print 10,text2
print 10,text3
10 format (A15)
```

erzeugt die Ausgabe:

```
Projektgruppe␣␣
Praktische␣␣␣␣␣
␣␣␣Mathematik␣␣
```

Erklärung: Wird beim A-Format die Anzahl der zu druckenden Stellen angegeben, gibt es zwei Möglichkeiten:

- *w* ist *kleiner* als die Länge der Charactervariablen. Dann werden die ersten *w* Zeichen der Charactergröße gedruckt.
- *w* ist *größer oder gleich* der Länge der Charactergröße. In diesem Fall wird die Charactergröße *rechtsbündig* in dem vorgesehenen Feld ausgegeben.

In dem Beispiel werden drei Charactervariablen mit unterschiedlicher Länge im A-Format A15 ausgegeben. In den beiden ersten Fällen `text1` und `text2` ist die Länge größer als 15 Zeichen, d.h. die ersten 15 Zeichen der Charactervariablen werden gedruckt. Im dritten Fall `text3` ist die vereinbarte Länge kleiner als die angegebene Breite des Ausgabefeldes. Die 12 Zeichen von `text3` werden deshalb in einem 15 Stellen breiten Feld *rechtsbündig* ausgegeben, der Rest wird mit Leerzeichen aufgefüllt. Unkomplizierter in der Anwendung ist das einfache A-Format ohne Angabe der Feldweite!

Das Beispiel zeigt übrigens auch, dass sich mehrere `print`-Anweisungen auf die gleiche `format`-Anweisung beziehen können.

In einer Ausgabeliste können natürlich verschiedene Datentypen gemischt auftreten. Es ist zu beachten, dass auch in der zugehörigen `format`-Anweisung entsprechende Formatbeschreiber in der gleichen Reihenfolge auftauchen. Beispiel:

```
integer :: i, j
real :: x, y
character ( len = 40) :: text
⋮
print 10, i, x, text, y, j
```

Die zugehörige `format`-Anweisung mit dem Label `10` könnte z.B. so aussehen:

```
10 format (I5, F10.5, A, E12.5, I3)
```

Die bis jetzt vorgestellten Formatbeschreiber beziehen sich alle auf Elemente der Ausgabeliste. Zusätzlich zu diesen Beschreibern können weitere Angaben in der Formatanweisung gemacht werden, um:

- die Position, an der gedruckt werden soll, festzulegen
- Text auszugeben

Der T-Formatbeschreiber zur Positionierung:

Durch die Angabe von:

T*n*

legt man fest, dass die Ausgabe in der *n*-ten Spalte einer Zeile fortgesetzt werden soll. Beispiel:

```
character ( len = 15) :: text1
real :: x
x = 10.5
text1 = 'Wert von x ='
print 20 , text1, x
20 format (T10, A, T30, F5.2)
```

erzeugt die Ausgabe:

␣␣␣␣␣␣␣␣␣Wert von x =␣␣␣ ␣␣␣␣␣10.50

Erklärung: Der erste Formatbeschreiber in der Formatanweisung lautet `T10`. Diese Beschreibung bezieht sich nicht auf die erste zu druckende Größe `text1`, sondern bewirkt, dass alles weitere ab der 10. Spalte ausgegeben wird. Der nächste Formatbeschreiber A bezieht sich auf das erste Element der Ausgabeliste. `text1` wird in einem 15 Spalten breiten Feld von Spalte 10 bis 24 ausgegeben (die kleine Lücke). Durch `T30` wird die Ausgabe in Spalte 30 fortgesetzt. `F5.2` bezieht sich auf das nächste Element der Ausgabeliste `xwert`. `xwert` wird in einem 5 Spalten breiten Feld von Spalte 30 bis 34 ausgegeben.

Die T-Anweisung ist besonders praktisch, wenn man übersichtliche Tabellen schreiben möchte.

Der X-Formatbeschreiber zur Positionierung:

Durch die Angabe von:

*n*X

werden *n*-Leerstellen ausgegeben. Wie beim T-Formatbeschreiber wird auch beim X-Beschreiber kein Element der Ausgabeliste abgearbeitet.

Beispiel:

```
print10, 1, 2, 3, 4, 5
10 format(1X, I1, 2X, I1, 3X, I1, 4X, I1, 5X, I1)
```

erzeugt die Ausgabe:

```
␣1␣␣2␣␣␣3␣␣␣␣4␣␣␣␣␣5
```

Erklärung: Alle `integer`-Konstanten werden im `I1`-Format ausgegeben, d.h. jeweils eine Stelle steht zur Darstellung zur Verfügung. Dazwischen werden Leerstellen in größer werdender Zahl ausgegeben. Das Beispiel erinnert auch noch einmal daran, dass in einer Ausgabeliste auch Konstante Zahlen stehen können.

Der Schrägstrich (/) zur Positionierung:

Durch einen Schrägstrich (/) wird die Ausgabe in einer Zeile abgeschlossen, alles weitere wird, beginnend am Anfang der nächsten Zeile, ausgegeben. Beispiel:

```
print 10, ' Alte Zeile',' Neue Zeile'
10 format (A,/,A)
```

erzeugt die Ausgabe:

```
Alte Zeile
Neue Zeile
```

Erklärung: Die Characterkonstanten werden beide im A-Format ausgegeben, d.h. die Zahl der ausgegebenen Stellen entspricht der Länge der Konstanten. Der Schrägstrich bewirkt einen Sprung an den Anfang einer neuen Zeile, nachdem die erste Konstante ausgegeben wurde.

Durch n aufeinanderfolgende Schrägstriche in einer Formatanweisung erzeugt man $n-1$ Leerzeilen in der Ausgabe! In Tabelle 5 sind zusammenfassend die bisher erläuterten Formatbeschreiber aufgelistet.

Formatbeschreiber	**Anwendung**
I*w*	Für `integer`-Größen
F*w.d*	Für `real`, `double precision` und `complex`-Größen
E*w.d*	Für `real`, `double precision` und `complex`-Größen
A[*w*]	Für `character`-Größen
T*n*	Positionierung in der *n*-ten Spalte
*n*X	Ausgabe von *n* Leerstellen
/	Zeilenende (Ausgabe in der nächsten Zeile)

(*w* $\hat{=}$ Gesamtzahl Stellen, *d* $\hat{=}$ Nachkommastellen)

Tabelle 5: Formatbeschreiber für die Ausgabe von Variablen und Konstanten

Ausgabe von Text in der Formatanweisung:

Es ist erlaubt, in der Formatanweisung `character`-Konstante anzugeben. Diese werden mit ausgegeben. Beispiel:

```
print 10,' Adam','Eva'
print 10,' Max','Moritz'
10 format (A,' und ',A)
```

erzeugt die Ausgabe:

```
Adam und Eva
Max und Moritz
```

Erklärung: Im ersten `print`-Befehl werden die Characterkonstanten `Adam` und `Eva` aufgeführt. Ihre Ausgabe wird in der Formatanweisung durch das A-Format definiert. In der Formatanweisung selbst ist zwischen die beiden A-Formatbeschreiber die Characterkonstante `und` eingefügt, die mit ausgegeben wird.

Diese Art der Ausgabe von Text wird oft dann verwendet, wenn verschiedene Variablen immer wieder mit dem gleichen Text kombiniert ausgegeben werden müssen. Der Text muss dann nicht mehr im `print`-Befehl aufgeführt werden, sondern erscheint nur noch in der `format`-Anweisung. Verschiedene `print`-Befehle beziehen sich dann auf diese `format`-Anweisung.

Wiederholungen von Datenformatbeschreibern:

Alle Formatbeschreiber können mit Wiederholungsfaktoren versehen werden, z.B. statt:

```
10 format (F5.2, F5.2, F5.2)
```

schreibt man:

```
10 format (3 F5.2)
```

Auch Kombinationen von Beschreibern können so behandelt werden, z.B.:

```
10 format (F5.2, 2X, F5.2, 2X, F5.2, 2X)
```

durch:

```
10 format (3 (F5.2, 2X))
```

Beispiel:

Das Programmsegment

```
character (len = 40) :: l1
character (len = 40) :: l2
l1='i-------------------------------------------------------i'
l2='i-------------------------------------------------------i'
print20,l1, l2, l2, l2, l1, l2, l2, l2, l1, l2, l2, l2, l1
20 format (13(T10,A40,/))
```

erzeugt als Ausgabe ein kleines Schachbrett.

8.2 Formatsteuerung bei der Eingabe

Bei der Eingabe von Daten mit Hilfe der Anweisung

```
read * , v [, v] ...
```

wurden durch Trennzeichen verschiedene Werte über eine Eingabeliste den Variablen zugeordnet. Trennzeichen sind das Komma, ein oder mehrere Leerzeichen oder ein Schrägstrich (/). Auch eine neue Zeile hat die Funktion eines Trennzeichens.

Wie beim `print`-Befehl erlaubt Fortran95 auch beim `read`-Befehl die Definition von eigenen Formatbeschreibern. Die Eingabe wird dann in einem genau definierten Feld einer Eingabezeile erwartet. Die Formatbeschreiber sind die gleichen wie bei der Ausgabe (siehe Tabelle 5).

Die Verwendung von Formatanweisungen in der Eingabe kann an einigen Beispielen gezeigt werden:

`I`-Format bei der Eingabe:

Beispiel:

```
integer :: i, j
read 10, i, j
10 format (I5,I7)
```

Der Wert der Variablen `i` wird im Feld 1-5 erwartet, der Wert der Variablen `j` im Feld 6-12. Die nicht besetzten Stellen werden mit Nullen aufgefüllt.

F- und E-Format bei der Eingabe:

F-Format und E-Format haben bei der Eingabe identische Wirkung! Für F könnte also auch immer E stehen in den folgenden Beispielen. Durch die Angabe der Feldweite wird bei F-Format ein Bereich definiert, in dem die Zahl gelesen wird. Beispiel:

```
real :: var
read 10, var
10 format (F10.3)
```

Um der Variablen `var` durch diesen Programmteil den Wert -8.45 zuzuweisen, haben wir mehrere Möglichkeiten. Grundsätzlich wird der Wert in dem Bereich von der ersten bis zur zehnten Spalte einer Zeile erwartet:

Eingabeformat	Bemerkung
`-8.45`	Darstellung mit Dezimalpunkt
`-.845E1`	Exponentialdarstellung mit E
`-84.5-1`	Exponentialdarstellung ohne E
`-8450`	Darstellung ohne Dezimalpunkt
`-845E1`	Exponentialdarstellung ohne Dezimalpunkt

Es ist zulässig, den Dezimalpunkt bei der Eingabe von `real`-Zahlen wegzulassen. Die letzten *d* Stellen der Zahl werden als Nachkommastellen interpretiert, die Integerzahl *d* wird in der Formatanweisung definiert. Im Beispiel ist $d = 3$ wegen der Formatanweisung `F10.3` sodass die letzten drei Stellen der Zahl als Nachkommastellen interpretiert werden. Die Stellung der Zahl innerhalb des Bereichs der Spalten 1 bis 10 spielt *keine* Rolle.

Es wird empfohlen, bei der Eingabe von `real`-Zahlen nur die Darstellung als Zahl mit Dezimalpunkt oder in Exponentialdarstellung mit dem Buchstaben `E` und mit Dezimalpunkt zu verwenden.

A-Format bei der Eingabe:

Wie bei der Ausgabe, sieht der formale `A`-Formatbeschreiber folgendermaßen aus:

`A[`*w*`]`

Wird der Formatbeschreiber `A` verwendet, stimmt die Länge des Eingabebereichs mit der Länge der Charactervariablen überein. Wird der Formatbeschreiber A*w* verwendet, gelten folgende Regeln:

- *w* ist *kleiner* als die Länge der Charactervariablen. Dann werden *w* Zeichen gelesen und der Rest der Charactervariablen wird mit *len*-*w* Leerzeichen aufgefüllt. (*len* ist die Länge)
- *w* ist *größer oder gleich* der Länge. Dann werden die am weitesten *rechts* stehenden *len* Zeichen aus dem Eingabefeld entnommen.

Formatbeschreiber zur Positionierung:

Die bei der Ausgabe vorgestellten Formatbeschreiber zur Positionierung `T`, `X` und `/` wirken bei der Eingabe in gleicher Weise. Dadurch ist es zum Beispiel möglich, erst ab Spalte 20 zu lesen oder ganze Zeilen zu überlesen.

Variable Formatangaben:

Bei der Standard-Ausgabeformatierung von Fortran95 ist man darauf angewiesen, dass das Ausgabeformat zum Zeitpunkt der Kompilation des Programmes bereits bekannt ist. Die Verwendung von Variablen in einer `format`-Anweisung ist nicht möglich. In Abschnitt 10 Variable Formatangaben ist ein Weg dargestellt, um variable Format-Angaben während des Programmablaufes zu konstruieren. Da dieses Verfahren auf die Verwendung von Dateioperationen aufbaut, werden diese zunächst im folgenden Abschnitt eingeführt.

9 Dateioperationen

9.1 Dateien Öffnen (open-Anweisung)

Bisher wurde die gesamte Daten Ein- und -Ausgabe über den Terminalbildschirm abgewickelt. Das ist für viele Anwendungsfälle sehr umständlich und unkomfortabel. Es gibt aber die Möglichkeit, auch von Dateien zu lesen und auf Dateien zu schreiben.

Vor jedem Zugriff auf eine Datei von einem Fortran95-Programm aus muss sie im Programm *geöffnet* werden. Das geschieht durch eine `open`-Anweisung. Durch die `open`-Anweisung wird einer Datei eine Dateinummer zugeordnet, unter der sie anschließend angesprochen wird. Allgemein hat die `open`-Anweisung die Form:

```
open (oliste)
```

Die *oliste* enthält Angaben über die Datei, über die Art des Zugriffs auf die Daten der Datei, sowie Anweisungen, was geschehen soll, wenn Fehler auftreten. In diesem Kurs werden nur die wichtigsten und notwendigen Angaben erwähnt. Die einfachste Form der `open`-Anweisung ist

```
open (unit = u, file = dateiname [, iostat = iostat] [, status = status])
```

dabei bedeutet *u* die Dateinummer, die man sich selbst ausdenkt ($7 \leq u < 99$) und *dateiname* ist ein Characterausdruck, der den Namen der Datei angibt, welche im folgenden unter der Nummer *u* angesprochen werden kann. *iostat* ist eine ganzzahlige Statusvariable, der der Wert Null zugewiesen wird, wenn beim öffnen der Datei kein Fehler aufgetreten ist. Der *status*-Ausdruck spezifiziert den Status der Datei. Hier gibt es z.B. `old` bei dem die Datei schon existiert, `new`, die Datei existiert noch nicht und `unknown` welches die voreingestellte Form ist, die betriebssystemabhängig ist.

Beispiele, um Dateien zu öffnen:

```
open (unit = 7, file = 'Eingabedaten')
```

oder:

```
character (len = 12) :: text
text='ausgabe'
open (unit = 9, file = text)
```

Input/Output- Status (`iostat`):

Die *iostat*-Variable nimmt einen ganzzahligen Wert ungleich 0 an, wenn beim Ausführen der Anweisung ein Fehler auftrat. Dieser Wert ist eine betriebssystemspezifische Fehlernummer, zu der eine bestimmte Fehlermeldung gehört. Die Variable wird hier mit `ios` bezeichnet. `ios` kann folgende Werte annehmen:

Wert	Status
`ios<0`	Das Dateiende ist erreicht
`ios=0`	o.k.
`ios>0`	Fehler

Die Fehlermeldung kann mit einer `subroutine` abgefragt werden, die vom System zur Verfügung gestellt wird.

Beispiel, um Dateien zu öffnen und dabei auf Fehler abzufragen:

```
subroutine aktiv
    ....

    ! Datei oeffen
    open (unit=20,file='preise.dat',iostat=ios,status='old')

    ! Falls beim Oeffnen Fehler auftritt, Fehlerausgabe und Abbruch
    if(ios/=0) then
        print*,'Datei nicht vorhanden oder fehlerhaft'
        ! Sprung aus der subroutine
        return
    end if
    ....
end subroutine aktiv
```

Hierbei muss folgendes beachtet werden. Wenn man für status 'old' angibt, was heißt, dass man davon ausgeht, dass die Datei schon existieren soll, dann gibt es für den Fall, dass die Datei noch nicht existiert einen Fehler, d.h. ios wird einen Wert ungleich Null erhalten. Wählt man hingegen 'new', so wird die Datei bei Nichtvorhandensein angelegt, ohne dass es einen Fehler gibt, d.h. ios wird zu Null.

9.2 Dateien Lesen und Schreiben (`read`- und `write`-Anweisung)

Nach dem öffnen von Dateien können diese in nachfolgenden Ein- und Ausgabeanweisungen angesprochen werden. Im Unterschied zu der bisher verwendeten `read`- und `print`-Anweisung muss dazu auch die Dateinummer mit angegeben werden. Zusätzlich kann man weitere Angaben machen, was geschehen soll, wenn z.B. Fehler beim Lesen auftreten. Wie bei der `open`-Anweisung sollen zuerst Beispiele für die einfachste Form des Lesens und Schreibens dargestellt werden und danach Möglichkeiten zum Fehlerabfangen. Die einfachste Form der `read`-Anweisung mit Zugriff auf eine vorher geöffnete Datei lautet

```
read (unit = u, fmt = f)
```

u ist die Dateinummer der Datei, von der gelesen werden soll, *f* ist das Label einer Formatbezeichung, z.B. `10`, oder auch ein `*`, wenn `listengesteuert` eingelesen wird. Beispiel:

```
real :: x
open (unit = 7, file = 'Einlesedatei')
read (unit = 7, fmt = 10) x
10 format (F20.10)
```

Dieses Programmstück bewirkt, dass eine Zahl in der ersten Zeile der Datei `Einlesedatei` in einem Bereich von Spalte 1 bis 20 gelesen und der `real`-Variablen `x` zugeordnet wird. Dabei sind die Klammern zu beachten und dass hinter der schließenden Klammer *kein* Komma steht!

Nun ein Beispiel mit Fehlerabfangen:

```
....
! Preise fuer den Stahl einlesen
do i=1,20
    read(unit=20, fmt=*,iostat=ios)stahlpreise(i)

    ! Fehlerbehandlung
    if(ios/=0) then                ! falls Fehler
        print*,'Datei fehlerhaft'  ! Fehlermeldung
        return                     ! und Abbruch
    end if
end do
....
```

Für das Schreiben auf Dateien kann man den `print`-Befehl *nicht* verwenden. Er wird ersetzt durch die `write`-Anweisung, die in der einfachen Form so aussieht:

```
write (unit = u, file = f)
```

Wie bei der `read`-Anweisung soll auch bei der `write`-Anweisung nicht auf die möglichen zusätzlichen Angaben eingegangen werden. Das Beispiel zeigt, wie man Text auf eine Datei schreiben kann:

```
open (unit = 9, file = 'Ausgabedatei')
write (unit = 9, fmt = 20)
20 format ('Das ist die Datei Ausgabedatei!!')
```

Nachdem das Programm gelaufen ist, steht in der ersten Zeile der Datei `Ausgabedatei` der gewünschte Text.

Beispiel:

Es soll ein Programm geschrieben werden, das den Nachnamen von der ersten Zeile und den Vornamen von der zweiten Zeile der Datei `eingabe` lesen soll. Dann soll `vorname` und `nachname` in die erste Zeile der Datei `ausgabe` geschrieben werden:

```
program name
implicit none
character (len = 20) :: nachname
character (len = 20) :: vorname
open (unit = 7, file = 'eingabe')
open (unit = 9, file = 'ausgabe')

read (unit = 7, fmt = *) nachname
read (unit = 7, fmt = *) vorname

write (unit = 9, fmt = *) vorname, nachname

end program name
```

Eine Datei, von der gelesen werden soll, muss vor dem Rechenlauf mit (sinnvollen) Daten beschrieben sein. Eine Datei, auf die nur geschrieben wird, muss nicht vorher angelegt werden. Durch die `open`-Anweisung wird sie erzeugt und existiert nach einem Lauf des Programms.

Gelesen und geschrieben wird immer von der ersten Zeile einer Datei beginnend.

9.3 Dateien Schließen

Wird in einem Programm der `end`-Befehl ausgeführt, wird es „kontrolliert“ abgebrochen, d.h. dass spätestens zu diesem Zeitpunkt alle Daten auf die geöffneten Dateien geschrieben, und dass sie geschlossen werden. Damit sind die Dateien unter den Namen, unter denen sie geöffnet wurden, allgemein verfügbar. Solange sie durch `open` geöffnet sind, ist ein Zugriff auf diese Dateien nur dem Programm möglich.

Wenn ein Programm unkontrolliert, z.B. durch Programmabsturz, abgebrochen wird, sind die Daten häufig noch nicht auf die Dateien übertragen. Um dies jedoch im Programm sicherzustellen, sollte die `close`-Anweisung verwendet werden. Durch `close` wird eine geöffnete Datei geschlossen.

```
close (unit = u [, iostat = iostat] [, status = status])
```

Für *u* und *iostat* gilt das gleiche, wie bei der `open`-Anweisung.

status ist ein String, der die Werte `keep` oder `delete` annehmen kann. Wird *status* nicht angegeben, wird der Wert `delete` angenommen. Ist der Status `keep`, heißt das, dass die Datei noch benötigt wird und weiterhin nur vom Programm auf sie zugegriffen werden kann. Dies bleibt solange der Fall, bis ein `close` mit `delete` oder ein Programmabbruch erfolgt. Ist der Status `delete`, wird die Datei „zurückgegeben“ und steht allgemein zur Verfügung.

9.4 Die `rewind`-Anweisung

Durch die `rewind` -Anweisung ist es möglich, im Programm eine Datei auf den Anfang zu positionieren um Daten nochmals einzulesen. Das folgende Beispiel zeigt einen sinnvollen Anwendungsfall.

Beispiel:

Aus einer Reihe von Zahlen soll die betragsmäßig größte herausgesucht werden. Dann soll das prozentuale Verhältnis der übrigen zu der größten berechnet und ausgegeben werden. Da die Anzahl der Zahlen nicht bekannt ist, ist ein Vorgehen mit Vereinbarung eines Feldes ungünstig. Das Problem kann gelöst werden mit Hilfe der `rewind`- Anweisung:

```
program zahlen
implicit none
⋮
open (unit = 8, file = 'eingabe')

! Im ersten Durchlauf wird nur die größte Zahl gesucht.
! Die Anzahl gibt die Anzahl der Zahlen an.
anzahl = 100
zahlmax = 0.
do i = 1, anzahl
  read (unit = 8, fmt = *) zahl
     if (zahl > zahlmax)
      zahlmax = zahl
end do

! Im zweiten Durchlauf werden Werte berechnet
 rewind (unit = 8)

 do i = 1 , anzahl
   read (unit = 8,fmt = *) zahl
   proz = zahl / zahlmax * 100.
   print*, zahl, proz
 end do

 close (unit = 8)
 end program zahlen
```

10 Variable Formatangaben

Wer bereits andere Programmiersprachen kennt und benutzt hat, wird sich vielleicht schon gefragt haben, warum es mit Fortran so ausgesprochen kompliziert ist, optisch ansprechende Ausgaben von Zahlen und Zeichenketten zu erzeugen oder Daten aus Dateien zu lesen, bei denen Anzahl und Formatierung nicht von vornherein exakt festgelegt sind.

Die Antwort auf diese Frage kennen wohl nur die Programmierer, die Fortran ursprünglich entwickelt haben. Auch Fortran95 hat diese umständliche Ein- und Ausgabesteuerung aus Kompatibilitätsgründen geerbt, aber diese Version bietet einige Erweiterungen, mit denen man variable Formatangaben während der Laufzeit eines Programmes erzeugen kann.

Schritt 1: Zeichenketten als Formatbeschreiber

Bei der `fmt`-Option von `read`- und `write`-Anweisungen ist es statt der Angabe eines Format-Labels auch möglich, eine Zeichenkette, bzw. Zeichenkettenvariable anzugeben. Weiterhin kann bei der `unit`-Option statt der Dateinummer einer zuvor geöffneten Datei auch ein Sternchen („`*`“) angegeben werden, der das Standardausgabe-Gerät (Bildschirm), bzw. das Standardeingabe-Gerät (Tastatur) bezeichnet.

Im folgenden Beispiel wird dreimal die gleiche Ausgabe mit unterschiedlichen Methoden erzeugt:

```
character*20 :: formatzk
real :: a
⋮
a=20.1253

! klassische Formatangabe:
print 10,'Der Wert von a beträgt: ',a
10 format (A,F4.1)

! Ausgabe der Zeile mit einer Zeichenkette als Formatbeschreiber:
write(unit=*,fmt='(A,F4.1)') 'Der Wert von a beträgt: ',a

! Ausgabe der Zeile mit einer Zeichenkettenvariable:
formatzk='(A,F4.1)'
write(unit=*,fmt=formatzk) 'Der Wert von a beträgt: ',a
```

Mit dieser Methode kann man sich nun zunächst einmal von der fest einprogrammierten und unveränderlichen `format`-Anweisung trennen. Es ist damit prinzipiell auch schon möglich, variable Formatbeschreiber zu erzeugen, indem man beispielsweise den Benutzer im Programm auffordert, einen passenden Format-Beschreiber einzugeben oder indem man in die erste Zeile einzulesender Dateien einen für diese Datei passenden Formatbeschreiber einfügt, der dann vom Programm eingelesen und verwendet wird.

Schritt 2: Zeichenkettenvariablen als schreib- und lesbare Dateien

Im ersten Schritt wurde die `format`-Anweisung durch die Verwendung einer Zeichenkette ersetzt. Im nächsten Schritt soll nun eine numerische Variable in eine Zeichenkette umgewandelt werden

Mit Fortran95 ist es möglich, Zeichenkettenvariablen wie Dateien zu behandeln, d.h. mit einer `read`-Anweisung können Werte aus einer Zeichenkette gelesen werden und z.B. als numerische Variable gespeichert werden und mit `write` können umgekehrt numerische Werte in lesbarer Form in Zeichenkettenvariablen geschrieben werden. Für das Lesen und Schreiben kann, wie auch bei „echten“ Dateien,

ein Formatbeschreiber angegeben werden. *Dieser ist aber nicht zu verwechseln mit dem Formatbeschreiber in Form einer Zeichenkette, der im dritten Schritt erzeugt werden soll!* Beispiel:

```
character z*4
integer :: i
⋮
! Lesen einer Integer-Zahl (formatfrei)
! aus der Zeichenkettenvariable z:
z='123'
read(unit=z,fmt=*) i
print *,i              ! Ausgabe: ' 123'

! Schreiben einer Integer-Zahl (dreistellig)
! in die Zeichenkettenvariable z:
i=456
write(unit=z,fmt='(I3)') i
print *,z              ! Ausgabe: ' 456'
```

Hinweis:

Dieses Verfahren kann auch ganz allgemein für die **Typenumwandlung** numerischer Variablen in Zeichenketten und umgekehrt angewendet werden.

Schritt 3: Erzeugen eines variablen Formatbeschreibers

Im zweiten Schritt wurde gezeigt, wie eine numerische Angabe in eine Zeichenkette gewandelt werden kann. Jetzt soll ein vollständiger Formatbeschreiber erzeugt werden, in den auch Werte numerischer Variablen eingebaut werden. Im folgenden Beispiel wird der Benutzer gebeten, für die Ausgabe von a (siehe Beispiel im 1. Schritt) die gesamte Anzahl der Stellen sowie die Nachkommastellen anzugeben:

```
character*20 :: formatzk
integer :: w,d
real :: a
⋮
a=20.1253

print *,'Gesamtanzahl der Stellen: '
read *,w
print *,'Nachkommastellen: '
read *,d

write(unit=formatzk,fmt=*) '(A,F',w,'.',d,')'

print *
print *,'Erzeugter Formatbeschreiber: ',formatzk
print *
write(unit=*,fmt=formatzk) 'Der Wert von a beträgt: ',a
```

Bei Ausführung dieses Programmstücks stellt man fest, dass der hiermit erzeugte Formatbeschreiber jeweils Leerzeichen vor den einzelnen Zahlenangaben enthält, da die Zeichenkette selbst wiederum formatfrei erzeugt wurde („`fmt=*`“). Das stört bei den meisten Formatbeschreibungen auch nicht. Will

oder muss man diese Leerzeichen aber unterbinden, muss für die *Erzeugung* des Formatbeschreibers ebenfalls noch ein Format angegeben werden:

```
write(unit=formatzk,fmt='(A,I1,A,I1,A)') '(A,F',w,'.',d,')'
```

11 Benutzerdefinierte Datentypen

Bisher waren die verwendeten Daten immer vom gleichen Typ: Ganzzahlen (`integer`), Gleitpunktzahlen (`real` oder `double precision`) oder Zeichen (`character`). Das galt auch, wenn gleichartige Größen zu Feldern zusammengefasst wurden, z.B. Vektoren, Matrizen oder Strings (Zeichenketten).

Für manche Problemstellungen ist es aber sinnvoll und bequem, Daten unterschiedlichen Typs zu einem „**Dingsda**“ zusammenzufassen, wie im folgenden Beispiel:

In einer Galerie[23] sollen alle Exponate per EDV erfasst werden. Jedes Ausstellungsstück erhält eine Nummer, unter der verschiedene Informationen abrufbar sein sollen:

- der Titel des Werkes (Zeichenkette - `character`)
- der Name des Künstlers (Zeichenkette - `character`)
- das Entstehungsjahr (Ganzzahl - `integer`)
- der angestrebte Verkaufspreis (in Euro - `real`)
- die vereinbarte Provision (in Prozent - `integer`)

Diese Daten unterschiedlichen Typs können mit einer **`type`-Anweisung** zu einer sog. **Datenstruktur** (oder kürzer: **Struktur**) zusammengefasst werden (aus Gründen, deren Erläuterung hier zu weit führen würde, muss das an sich optionale **Attribut `sequence`** immer vorhanden sein):

```
type exponat_dat
   sequence
   character (len = 50) titel
   character (len = 50) name
   integer jahr
   real preis
   integer provision
end type exponat_dat
```

Dadurch wird ein neuer Datentyp namens `exponat_dat` *definiert*. Dieser neue Datentyp kann nun wie die vordefinierten Datentypen in einem Programm verwendet werden und wird mit dem Schlüsselwort **`type`** wie folgt zu Beginn des Programms vereinbart:

```
program galerie
implicit none
type exponat_dat                    ! Definition des neuen Datentyps
   sequence
   character (len = 50) titel
   character (len = 50) name
   integer jahr
   real preis
   integer provision
end type exponat_dat
integer :: irgend_ne_ganze_zahl     ! wie gehabt: integer
real :: irgend_ne_reelle_zahl       ! wie gehabt: real
type( exponat_dat ) :: kunstobjekt  ! neu: Vereinbarung der Variablen
⋮
```

[23] Dieses Beispiel wurde speziell für Teilnehmer des FB 1 (die Künstler) entwickelt!

Unter dem Variablennamen `kunstobjekt` können jetzt alle o.g. Daten unter einem gemeinsamen Namen angesprochen werden. Das kann z.B. verwendet werden, um einer Unterfunktion statt mehrerer einzelner Parameter nur eine entsprechende Datenstruktur zu übergeben, die „im Paket“ alle Daten enthält (vgl. Abschnitt 11.1 Übergabe von Strukturen an Unterprogramme). Ihren vollen Nutzen entfalten die `benutzerdefinierten Datenstrukturen` allerdings erst dann, wenn sie in Feldern zusammengefasst werden (s. Abschnitt 11.2 Arrays von Strukturen).

Zunächst sollen den Strukturkomponenten aber Werte zugewiesen werden, was auf zwei Arten möglich ist. Ist der Inhalt der Komponenten konstant, so können die Werte in Form einer Liste folgendermaßen zugewiesen werden:

```
kunstobjekt=exponat_dat('Unix-Pool in Öl', 'Marc Setzer', 2001, 1000, 50)
```

Formal sieht diese Art der Zuweisung so aus, dass der Name des benutzerdefinierten Datentyps `exponat_dat` wie eine Funktion verwendet wird, und die aktuellen Parameter werden den entsprechenden Komponenten zugewiesen.

Anstatt allen **Strukturkomponenten** in Form einer Liste auf einmal einen Wert zuzuweisen, können einzelne Komponenten mit dem **%-Operator**, der den Namen der Strukturvariablen und die Strukturkomponente trennt, gezielt angesprochen werden:

```
⋮
kunstobjekt%titel =  'Unix-Pool in Öl - impressionistisch'
kunstobjekt % name = 'Marc Setzer'
⋮
```

Wie man sieht, kann der %-Operator auch von Leerzeichen umgeben sein.

Die derart aus der Datenstruktur herausgelösten Komponenten werden wie normale Variable des entsprechenden Typs behandelt und können z.B. berechnet oder eingelesen werden (von der Tastatur oder aus einer Datei):

```
⋮
print * , 'Name des Werkes: '
read * , kunstobjekt%titel

print * , 'Name des Künstlers: '
read * , kunstobjekt%name
⋮
```

Zusammenfassung:

Die allgemeine Form der `Definition` einer benutzerdefinierten Datenstruktur lautet:

```
type typname
   sequence
   Datentyp_1 Komponente_1
   Datentyp_2 Komponente_2
   ⋮
end type [typname]
```

Datenstrukturen werden wie folgt zu Beginn einer Programmeinheit vereinbart:

```
type (typname) [, Attribute] :: Strukturvar_1 [, Strukturvar_2] ...
```

Auf einzelne **Strukturkomponenten** wird mit Hilfe des %-Operators zugriffen:

```
Strukturvariable%Strukturkomponente
```

Abschließend sei noch darauf hingewiesen, dass Datenstrukturen als Komponenten wiederum Strukturen enthalten können. Derartige Konstrukte tragen allerdings nicht unbedingt zur Übersichtlichkeit eines Programmes bei und sollten im Normalfall nicht erforderlich sein.

11.1 Übergabe von Strukturen an Unterprogramme

Die Übergabe einer benutzerdefinierten Datenstruktur an eine Funktion oder eine Subroutine verläuft im wesentlichen genauso wie die Übergabe einer Variablen vordefinierten Typs. In der aufrufenden Programmeinheit (meist dem Hauptprogramm, wie im folgenden Beispiel) wird der Name der Strukturvariablen als Aktualparameter übergeben:

```
⋮
type (exponat_dat) :: kunstobjekt ! Vereinbarung d. Variablen
                                  ! kunstobjekt v. Typ exponat_dat
⋮
call ausgabe( kunstobjekt )       ! Aufruf d. Subroutine "ausgabe()"
                                  ! mit Aktualparameter "kunstobjekt"
⋮
```

Unterschiede zum bisherigen Vorgehen gibt es nur in der aufgerufenen (Unter-) Programmeinheit: die Subroutine oder die Funktion kennt den neuen benutzerdefinierten Datentyp nicht. Die gleiche `type`-Anweisung wie im Hauptprogramm muss daher noch einmal wiederholt werden, und zwar *exakt*[24] mit den gleichen Bezeichnungen. Anschließend wird die Strukturvariable wie alle anderen Formalparameter vereinbart.

Ein Beispiel für eine komplette Subroutine:

```
subroutine ausgabe( objekt )
implicit none

type exponat_dat               ! benutzerdefinierter Typ
   sequence
   character (len = 50) titel
   character (len = 50) name
   integer jahr
   real preis
   integer provision
end type exponat_dat

type( exponat_dat ) :: objekt  ! Formalparameter vereinbaren
⋮
```

[24] Darauf sollte Sorgfalt verwandt werden!

```
 ⋮
print *
print * , 'Titel: ' , trim( objekt%titel )[25]
print * , 'erschaffen von: ' , trim( objekt%name )
print * , 'Provision:' , objekt%preis * objekt%provision / 100.

end subroutine ausgabe
```

Der hier beschriebene Umgang mit Datenstrukturen und Unterprogrammen bezieht sich auf *externe*[26] Unterprogramme, welche im Abschnitt 6 Prozedurale Programmierung und Unterprogramme behandelt wurden.

11.2 Arrays von Strukturen

Bisher kann lediglich *ein* Kunstgegenstand datentechnisch erfasst werden. Dafür lohnt die Anschaffung einer EDV-Anlage eigentlich nicht. Daher werden jetzt mehrere Strukturvariablen gleichen Typs zu **Feldern** zusammengefasst: jedes Feldelement ist ein benutzerdefiniertes Objekt.

Diese Konstruktion lässt sich gut mit einer herkömmlichen (Papier-) Kartei vergleichen:

- jede Karteikarte enthält Einträge für den Titel des Werkes, den Erschaffer, das Entstehungsjahr, den Verkaufspreis und die zu erzielende Provision – das ist die Datenstruktur `exponat_dat`;
- diese Karteikarten (so viele wie Ausstellungsstücke) werden sortiert in einer Pappschachtel aufbewahrt – das ist das Feld, bestehend aus Datenstrukturen.

Sofern die Datenstruktur einmal definiert worden ist, besteht kein Unterschied zur Vereinbarung eines „normalen“ Feldes (s. Abschnitt 7 Felder), das aus vordefinierten Typen besteht:

```
⋮
implicit none

type exponat_dat               ! benutzerdef. Typ
   sequence
   character (len = 50) titel
   character (len = 50) name
   integer jahr
   real preis
   integer provision
end type exponat_dat

double precision ,  dimension( 20 ) :: irgendein_feld ! wie vor
type (exponat_dat), dimension( 20 ) :: kartei         ! neu
⋮
```

[25] Die Funktion `trim()` löscht überflüssige Leerzeichen am Ende einer Zeichenkette
[26] Es gibt außerdem noch *interne* Unterprogramme und *Modul*-Unterprogramme, die in diesem Umdruck nicht behandelt werden.

Im obigen Beispiel wurde praktisch eine „Kartei“ mit 20 (leeren) Karteikarten angelegt. Die statische Dimensionierung mit einer festen Anzahl an Karteikarten ist noch nicht optimal, da sich die Anzahl der Ausstellungsstücke noch vergrößern kann. Zweckmäßigerweise wird der benötigte Speicher dynamisch alloziert (vgl. Abschnitt 7.4 Dynamische Speicherplatzverwaltung):

```
type (exponat_dat), dimension( : ), allocatable :: kartei
integer :: wieviele
⋮
read * , wieviele
allocate( kartei(wieviele) )
```

Dadurch werden `wieviele` (leere) Karteikarten vom Typ `exponat_dat` in der Kartei `kartei` angelegt.

Jedes Feldelement von `kartei` wird nun als Ganzes genauso behandelt wie eine einzelne Datenstruktur, z.B. bei der Übergabe an die Funktion `ausgabe`:

```
⋮
call ausgabe( kunstobjekt )  ! wie gehabt
call ausgabe( kartei(2) )    ! neu: 2. Element der Kartei ausgeben
⋮
```

Das obige Beispiel gibt mit Hilfe der Subroutine `ausgabe` zuerst die Komponenten (Karteikarteneinträge) der einzelnen (skalaren) Datenstruktur `kunstobjekt` und anschließend die Komponenten des Feldelementes `kartei(2)` aus.

Die Subroutine `ausgabe` (Seite 96) kann nur die Komponenten einer einzelnen Strukturvariablen von Typ `exponat_dat` (also immer nur eine Karteikarte) ausgeben, nicht aber die gesamte Kartei. Der Aufruf einer Subroutine, der als Parameter ein Feld aus Strukturvariablen übergeben wird, erfolgt genauso wie bei Feldern, die aus vordefinierten Datentypen bestehen (siehe hierzu auch Abschnitt 7.6), nämlich:

```
⋮
call ausgabe_alles( kartei , wieviele )  ! gesamte Kartei ausgeben
                                         ! variable Dimensionierung
⋮
```

Die neue Subroutine `ausgabe_alles` muss jetzt so geschrieben werden, dass sie ein Feld mit `wieviele` Elementen verarbeitet, wobei die Elemente Strukturobjekte vom Typ `exponat_dat` sind:

```
subroutine ausgabe_alles( kartei , anzahl )
implicit none

type exponat_dat               ! benutzerdefinierten Typ
   sequence                    ! in dieser Subroutine
   character (len = 50) titel  ! definieren/bekanntmachen
   character (len = 50) name
   integer jahr
   real preis
   integer provision
end type exponat_dat
⋮
```

```
⋮
! Formalparameter vereinbaren:
integer, intent( in ) :: anzahl    ! Länge des Feldes
type( exponat_dat ), dimension(anzahl) :: kartei
⋮
```

Um die gesamte Kartei auszugeben, ist nur noch die Frage zu klären, wie man an eine bestimmte Komponente eines Feldelementes „herankommt“. Das funktioniert im Prinzip genau so, wie bei skalaren Strukturobjekten: auf eine Komponente einer skalaren (einzelnen) Strukturvariablen wurde allgemein mit Hilfe des %-Operators zugegriffen:

```
Strukturvariable%Strukturkomponente
```

Jetzt ist die *Strukturvariable* ein Feldelement:

```
Strukturvariable (index)%Strukturkomponente
```

Es bleibt also alles beim Alten. Im folgenden wird die Subroutine `ausgabe_alles` von oben fortgesetzt. Alle Titel des Galerie-Bestandes werden der Reihe nach ausgegeben:

```
⋮
! lokale Schleifenvariable:

integer :: i

do i = 1 , anzahl
   print *
   print * , 'Titel: ' , trim( kartei(i)%titel)
   print * , 'erschaffen von: ' , trim( kartei(i)%name )
   print * , 'Provision:' , kartei(i)%preis * kartei(i)%provision / 100.
end do

end subroutine ausgabe_alles
```

Übung:

Jetzt sollten die Beispiel-Code-Schnipsel aus diesem Kapitel zu einem lauffähigen Programmchen zusammengefügt werden, um den Umgang mit Datenstrukturen etwas zu üben. Dabei empfiehlt es sich, nicht sklavisch alles abzutippen, sondern nach eigenem Gusto kleine Veränderungen oder Erweiterungen vorzunehmen.

12 Weitere Datentypen

12.1 Der Datentyp `complex`

Zum Umgang mit komplexen Zahlen gibt es den Datentyp `complex`. Bei diesem Datentyp wird eine komplexe Zahl

$$z = a + ib \qquad a, b \text{ reell} \qquad i = \text{imaginäre Einheit}$$

durch das reelle Zahlenpaar (a, b) repräsentiert.

Hier ein Beispiel, in dem die Diskriminante der quadratischen Gleichung $x^2 + (1.5 - 7.2i)x + (i + 1) = 0$ berechnet wird:

```
complex :: p, q, diskriminante
complex, parameter :: i=(0,1)
⋮
p=(1.5,-7.2)
q=i+1
diskriminante=p**2/4-q
⋮
```

Die Vereinbarung und Dimensionierung komplexer Felder erfolgt wie bei Feldern des `real`- oder `integer`-Typs. Doppelt genaue komplexe Daten werden mit der Anweisung **`double complex`** vereinbart.

Zwischen `complex`-, `real`- und `integer`-Variablen sind die Verknüpfungen `+`, `-`, `*`, `/` und `**` in jeder beliebigen Kombination erlaubt. Eine Verknüpfung einer Variablen vom Typ `real`, `integer` oder `complex` mit einer `complex`-Variablen ergibt immer einen Wert vom Typ `complex`. Darüber hinaus gibt es für `complex`-Variablen zahlreiche Standardfunktionen, von denen eine Auswahl in Tabelle 6 aufgeführt ist.

Ergebnis	Funktion	Parameter	Wirkung
komplex	`cmplx`	(x) reel	liefert komplexe Zahlendarstellung für x
reell	`real`	(x) komplex	liefert reelle Zahlendarstellung für x
reell	`aimag`	(x) komplex	liefert Imaginärteil von x
reel	`cabs`	(x) komplex	liefert Betrag von x
komplex	`conjg`	(x) komplex	liefert konjugiert von x
komplex	`csin`	(x) komplex	liefert $\sin(x)$ für x im Bogenmaß
komplex	`ccos`	(x) komplex	liefert $\cos(x)$ für x im Bogenmaß
komplex	`cexp`	(x) komplex	Exponentialfunktion zur Basis e
komplex	`clog`	(x) komplex	$\ln(x)$ Logarithmus zur Basis e, $x > 0$

Tabelle 6: Auswahl mathematischer Standardfunktionen für komplexe Zahlen

12.2 Der Datentyp `logical`

Daten vom Typ `logical` können nur die Werte **`.true.`** und **`.false.`** annehmen. Auf solche Daten kann man die in Tabelle 7 aufgeführten logischen Operatoren anwenden.

Operator	Bedeutung
`.not.`	Nicht
`.and.`	Und
`.or.`	einschließendes Oder
`.eqv.`	Äquivalenz (genau dann, wenn)
`.neqv.`	ausschließendes Oder (entweder.. oder)

Tabelle 7: logische Operatoren in Fortran95

Beispiele für Ausdrücke vom Typ `logical`:

1. `2<3`
2. `5>=7 .and. 2<3`
3. `5>=7 .or. 2<3`
4. `.not.( 5>=7.or.2*2>10) .or. 0<1`
5. `((-17>-18 .eqv. 6>7) .or. .not. (55>=4)) .and. .false.`

Der erste Ausdruck stellt eine wahre Aussage dar und hat daher den Wert `.true.`. Die zweite Aussage ist falsch und somit hat der zweite Ausdruck den Wert `.false.`. Der dritte und der vierte Ausdruck haben den Wert `.true.`. Der fünfte Ausdruck schließlich hat den Wert `.false.`, da das Ergebnis einer „und“-Verknüpfung immer eine falsche Aussage ergibt, wenn nur eine der beteiligten Aussagen falsch ist.

Übung:

Welchen Wert bekommt die Variable z durch folgenden Programmabschnitt:

```
logical :: x,y,z
⋮
x = .true.
y = (6*4>5*5)
z = (.not. y) .or. (.not. x)
⋮
```

Wichtig:

Will man eine Gleichung als logischen Ausdruck benutzen, so muss man ein **doppeltes Gleichheitszeichen** verwenden. Man betrachte z.B. die beiden Ausdrücke

```
n=n+1
```

und

```
n==n+1
```

Der erste Ausdruck `n=n+1` stellt eine Anweisung dar. Dieser Ausdruck bekommt den logischen Wert `.true.`, wenn die Anweisung erfolgreich durchgeführt wurde. Der zweite Ausdruck `n==n+1` stellt eine falsche Aussage (nämlich die mathematische *Gleichung* $n = n + 1$) dar und hat daher den Wert `.false.`.

Noch eine letzte Bemerkung: In einer `if`-Anweisung

```
if (Bedingung) then
   Anweisung
end if
```

ist die *Bedingung* vom Typ `logical`. Die *Anweisung* wird dann und nur dann ausgeführt, wenn die *Bedingung* den Wert `.true.` hat.

12.3 Der Datentyp `character`

Daten vom Typ `character` haben als Werte Zeichen oder **Zeichenketten**. Daten dieses Typs werden wie in folgendem Beispiel deklariert.

Beispiel:

```
⋮
character :: a, b, c, d*5
character*10 :: wort1, wort2
character, parameter :: klage='Worte, nichts als Worte'
character  ::  zfeld(10)*18
⋮
```

In der ersten Zeile des Beispiels werden zuerst die drei `character`-Variablen `a`, `b` und `c` deklariert, die jeweils nur aus einem Zeichen bestehen. Dann wird eine Variable `d` eingeführt, deren Werte Zeichenketten sind, die aus 5 Zeichen bestehen. In der zweiten Zeile werden die Variablen `wort1,wort2` deklariert, deren Werte Zeichenketten der Länge 10 sind. In der dritten Zeile wird die `character`-Konstante `klage` eingeführt. In der vierten Zeile wird ein Feld mit 10 Elementen eingeführt, die jeweils 18 Zeichen lang sind.

Noch ein Beispiel:

```
⋮
character :: a1*2, b1*4, a2*4, b2*2
⋮
b1 = 'beta'
a1 = b1
b2 = 'no'
a2 = b2
⋮
```

Nach Ausführung enthält `a1` die Zeichenkette „`be`“ und `a2` „`no␣␣`“. Es wird bei nicht übereinstimmender Länge also von rechts gekürzt bzw. mit Leerzeichen aufgefüllt.

Zeichenvariablen können in gewohnter Weise ausgegeben werden, Zeichenkonstanten werden in Apostrophe (’’) eingeschlossen. Beispiel – der Programmabschnitt:

```
⋮
character :: string*7
⋮
string = ’Ausgabe’
print *, ’ Dies ist eine’, ausgabe
⋮
```

liefert die Ausgabezeile

```
Dies ist eine Ausgabe
```

Bei der Eingabe sind Apostrophe unzulässig. So erwartet

```
⋮
character :: inzeil*15
⋮
read *,   inzeil
⋮
```

z.B. die Eingabezeile

```
Hallo Welt
```

Danach enthält `inzeil` den Wert „`Hallo␣Welt␣␣␣␣␣`“.

Es besteht auch die Möglichkeit, auf **Teile einer Zeichenkette** zuzugreifen:

```
⋮
character *6 ::  s1, s2, s3
⋮
s1 = ’string’
s2 = s1
s3 = s1(3:5)
⋮
```

Bei der Zuweisung von `s1` auf `s2` wird auf die komplette Zeichenkette in `s1` zugegriffen, während `s3` nach der Zuweisung den Wert „`rin␣␣␣`“ enthält.

Allgemein lautet ein Teilzeichenkettenzugriff:

```
name ([first] : [last])
```

`name` ist der Name einer `character`-Variablen.

`first` ist ein ganzzahliger Ausdruck, der die Position des ersten Zeichens der Teilzeichenkette angibt. Falls `first` fehlt, wird der Wert 1 angenommen.

`last` ist ein ganzzahliger Ausdruck, der die Position des letzten Zeichens der Teilzeichenkette angibt. Falls `last` fehlt, wird als Wert die Länge der Zeichenkette angenommen.

Weitere Beispiele:

`s1` wird, wie oben, der Inhalt `string` zugewiesen. Dann liefern die folgenden Teilzeichenketten die angegebenen Werte:

```
s1(1:3)        Wert: str
s1(3:4)        Wert: ri
s1(4: )        Wert: ing
s1( :4)        Wert: stri
s1( : )        Wert: string
```

Fortran95 stellt einen Zeichenoperator „`//`“ zur Verfügung, der zur Verkettung von Zeichenausdrücken dient. So liefert beispielsweise `'zeichen'//'kette'` den Wert „`zeichenkette`“. Durch die Verkettungsoperation wird also an einen Zeichenausdruck rechts ein anderer angebunden. Die Länge des Ergebnisses ist die Summe der Länge der Operanden. Als Operanden können natürlich auch Zeichenvariablen, Zeichenfeldelemente und Teilzeichenketten auftreten.

Sämtliche verfügbaren Zeichen sind durch den von dem Rechner benutzten Code (z.B. ASCII-Code) in ihrer Reihenfolge festgelegt. Durch die Rangfolge der Zeichen ist es möglich, mit Zeichen (-ketten) auch Vergleichsoperationen durchzuführen.

So liefert der Vergleichsausdruck in

```
⋮
integer :: st
character :: ind1, ind2
⋮
if (ind1 .lt. ind2) st = 15
⋮
```

nur dann den logischen Wert `.true.`, wenn das Zeichen in `ind1`, in der Anordnungsreihenfolge vor dem Zeichen in `ind2` steht. Dies gilt für längere Zeichenketten entsprechend („lexikalische Ordnung“).

In diesem Zusammenhang sind auch die **Standardfunktionen `char`** und **`ichar`** zu nennen. `char(n)` liefert für ein ganzzahliges Argument das Zeichen aus der Anordnungsreihenfolge mit der Nummer `n`. `ichar(z)` liefert für ein Argument vom Typ `character` (Länge 1!) die ganzzahlige Position dieses Zeichens innerhalb der Anordnungsreihenfolge.

Index