Fortran: eine ausführliche Minimal-Anleitung

Inhaltsverzeichnis

1 Warum Fortran? 3

2 Der Fortran-Compiler 4

2.1 Das Compiler-Konzept 4

2.2 Compiler-Optionen 4

3 Struktur eines Fortran-Programms 5

3.1 Allgemeiner Aufbau 5

3.2 Zeilenformat 6

3.2.1 festes Format 6

3.2.2 freies Format 6

4 Konstante und Variable 7

4.1 Grundkonzepte 7

4.2 Variablennamen 7

4.3 Variablentypen und -deklaration 8

4.3.1 Integervariable 8

4.3.2 real-Variable 8

4.3.3 Textvariable 9

4.4 implizite Typvereinbarung 9

4.5 Typumwandlung 10

4.5.1 automatisch 10

4.5.2 explizit 10

4.6 Felder 11

5 Rechenoperationen 11

5.1 Grundrechenarten mit Zahlen 11

5.2 Funktionen 12

6 Eingabe, Ausgabe und files 13

6.1 I/O-units 13

6.2 Öffnen von files 13

6.3 Schließen von files 14

6.4 Eingabe und Ausgabe 15

7 Kontrollstrukturen **15**

7.1 Bedingte Verzweigungen 16

7.1.1 allgemeine Struktur eines `if`-Blocks 16

7.1.2 Logische Operatoren und Bedingungen 16

7.2 Schleifen 17

7.2.1 do-while/until-Schleifen 18

7.2.2 Zählschleifen 18

7.3 Sprünge (zu vermeiden!) 19

8 Prozeduren **20**

8.1 Sinn von Prozeduren 20

8.2 Unterschied Funktion ↔ Unterprogramm 20

8.3 Aufruf und Rückkehr 21

8.4 Parameterlisten: einfache Variablen 22

8.5 Parameterlisten: Felder 23

8.6 Parameterlisten: optionale Argumente 26

8.7 Felder variabler Größe im Hauptprogramm 27

8.8 Verwendung von Bibliotheksroutinen 28

9 Programmierdisziplin! **30**

10 Wichtige, aber hier nicht behandelte Dinge **31**

Index **32**

1 Warum Fortran?

- höhere Programmiersprache mit längster Tradition
- seit Jahrzehnten totgesagt, aber trotz C (Unix/Linux!), C++, Java, ... immer noch sehr lebendig
- Programme klar und einfach lesbar
- für numerische Rechnungen am einfachsten in der Programmierung
- Compiler erzeugen dazu die schnellsten Programme
- daher nach wie vor der de-facto-Standard für „number crunching“ in Chemie, Physik, Computational Fluid Dynamics (Hydrodynamik, Aerodynamik, Wettervorhersage, Astronomie, usw.), ...
- sicher nicht die Sprache der Wahl für Textverarbeitung, Graphik, systemnahe Programmierung, usw.
- Neuerungen anderer Sprachen werden selektiv in Fortran übernommen
- derzeitiger Standard: Fortran 90 und Fortran 95 (der Fortran2003-Standard wurde Ende 2004 veröffentlicht; siehe homepages der Standard-Entwickler: `http://www.j3-fortran.org/` und `http://www.nag.co.uk/sc22wg5/`, aber dafür gibt es noch keine Compiler)
- wegen weitverbreiteter, großer software-Pakete, die der Standard-Entwicklung nicht so schnell folgen (können), sind alte Standards immer noch in aktivem Gebrauch, vor allem Fortran 77

Im Kurs wird der freie Fortran90/95-Compiler **gfortran** verwendet, der sowohl für Linux als auch für Windows verfügbar ist.

2 Der Fortran-Compiler

2.1 Das Compiler-Konzept

Der eigentliche Fortran-Programmtext kann mit einem beliebigen Texteditor erstellt werden (keine Formatierungsbefehle verwenden!). Er soll für den Programmierer verständlich sein, für den Computer ist er bedeutungslos. Es ist eine nützlichen Konvention, Fortran-Programme mit der Dateinamen-Erweiterung „.f90“ oder „.f95“ abzuspeichern. Der Hauptteil des Namens (vor dem Punkt) sollte den Programmierer an den Inhalt dieses Programms erinnern.

Der *compiler* übersetzt diesen Fortran-Text in eine direkt vom Computer ausführbare Form und legt das Resultat in eine neue Datei ab. Auf Windows-Systemen ist es nützlich, als Namen für diese Datei die Erweiterung „.exe“ zu wählen (unter Linux ist das unnötig/unüblich); der Hauptname (vor dem Punkt) sollte zur eigenen Orientierung identisch zu dem des zugehörigen „.f90/.f95“-files sein. Diese exe-Datei ist direkt ausführbar, und zwar ohne Anwesenheit des compilers (deshalb ist diese ausführbare Datei z.B. auch auf andere Computer (mit hinreichend kompatibler hardware und Betriebssystem) übertragbar und dort auch lauffähig).

Nach jeder Änderung des Programmtextes muß der compiler erneut gestartet werden, um diese Änderungen in die ausführbare Datei zu übernehmen; dies geschieht *nicht* automatisch.

2.2 Compiler-Optionen

Das Verhalten des compilers (und infolgedessen ggf. auch das Verhalten des ausführbaren Programms) kann durch compiler-Optionen beeinflußt werden, die beim compiler-Aufruf zusätzlich eingegeben werden. Die generelle Optionen-Syntax für `gfortran` ist die unter unix/linux übliche; Beispiel:

```
gfortran -Wall -o MeinProgramm MeinProgramm.f90
```

Alle Optionen beginnen mit einem Minuszeichen; sie stehen zwischen dem Befehl (hier: `gfortran`) und den eigentlichen Argumenten des Befehls (hier: `MeinProgramm.f`). Manche Optionen stehen für sich alleine (hier: `-Wall`). Andere brauchen ihrerseits Argumente, die dann direkt hinter der Option stehen müssen (wie hier: `-o MeinProgramm`).

Nützliche Optionen für gfortran:

`-help`: kurzer Hilfstext

`-Wall`: veranlaßt den compiler, soviele Warnhinweise auszugeben wie möglich.

`-pedantic`: warnt bei Programmkonstrukten, die nicht dem Standard entsprechen.

`-Wimplicit`: veranlaßt den compiler, eine Warnung auszugeben, wenn er im Programmtext auf Variablennamen stößt, die in keiner expliziten Typdeklaration vorkommen (nützlich zum Abfangen von Tippfehlern).

`-fimplicit-none`: äquivalent dazu, eine Zeile „`implicit none`“ in jeden Programmblock zu schreiben (ich empfehle, diese Zeilen trotzdem selber zu schreiben; diese Compiler-Option kann man leicht vergessen und dann sind die Konsequenzen in der Regel fatal!)

`-ffree-form, -ffixed-form`: Wie viele Compiler nimmt auch `gfortran` an, daß Dateien mit der Erweiterung „.f90“ bzw. „.f95“ im sogenannten freien Format vorliegen (siehe Abschnitt 3.2 auf S. 6) und Dateien mit der Erweiterung „.f“ im älteren festen Format. Mit diesen beiden Optionen kann die Format-Erwartung des Compilers unabhängig von dieser Grundannahme direkt eingestellt werden.

`-o name`: Das ausführbare Programm wird in der Datei „name“ gespeichert (beim Weglassen dieser Option ist das immer „a.out“).

`-O`: Optimierung; erzeugt schneller ausführbare Programme.

`-fbounds-check`: Überprüfung (und ggf. Warnhinweise oder Programmabbruch), wenn ein Feldindex die in der Vereinbarung des Feldes festgelegten Grenzen überschreitet. Bei anderen Compilern lautet diese Option üblicherweise `-C`.

`-g`: bereitet das erzeugte Programm für nachfolgende Bearbeitung mit dem debugger vor. Das resultierende Programm kann danach wie normal ausgeführt werden, aber auch innerhalb eines debuggers (z.B. mit `gdb MeinProgramm`).

3 Struktur eines Fortran-Programms

3.1 Allgemeiner Aufbau

Ein Fortran-Programm besteht aus direkt lesbarem Text, aus Buchstaben (zwischen Groß- und Kleinschreibung unterscheidet der compiler nicht!), Zahlen und einigen wenigen Sonderzeichen. Leerzeichen haben für Fortran-Compiler *keinerlei* Bedeutung; im Prinzip können sie sogar innerhalb von statement-Namen oder Variablennamen in beliebiger Zahl auftauchen, aber damit verwirrt man sich nur selber.

Im Allgemeinen steht in jeder einzelnen Programmzeile genau ein Befehl (Anweisung, command, statement). Diese Befehle werden bei Ausführung des Programms von oben nach unten nacheinander abgearbeitet (sofern nicht bestimmte Kontrollstrukturen diesen Ablauf modifizieren).

Bei einigen dieser Befehle „passiert tatsächlich etwas“ (es wird gerechnet, input oder output erzeugt, usw.); dies sind „ausführbare Befehle“. Andere Befehle dienen dazu, das Programm für den compiler zu strukturieren, Variablennamen zu vereinbaren, usw.; sie sind nicht im eigentlichen Sinne ausführbar.

Das kleinstmögliche Fortran-Programm hat deshalb mindestens zwei Zeilen, tut aber trotzdem gar nichts:

```
program minimal
end
```

An der ersten Zeile erkennt der compiler den Beginn eines (Haupt-)Programms, an der letzten Zeile das Ende eines Programmabschnitts (Hauptprogramm oder Unterprogramm).

Beim `program`-statement steht ein Name des folgenden Programmabschnitts; dieser Name dient zur Orientierung des Programmierers.

Der legale Aufbau eines nicht-trivialen Fortran-Programms sieht so aus:

Anfang: `program`-, `subroutine`- oder `function`-statement

Deklarationen: Vereinbarung von Variablen u.ä.

Ausführbarer Teil: alle ausführbaren Anweisungen

Ende: `end`-statement

Insbesondere dürfen Variablendeklarationen und ähnliche nicht-ausführbare Anweisungen nicht im ausführbaren Teil vorkommen und umgekehrt. Es gibt keine speziellen Grenzmarken zwischen diesen Abschnitten; z.B. beginnt der ausführbare Teil schlicht mit der ersten ausführbaren Anweisung.

3.2 Zeilenformat

3.2.1 festes Format

Im inzwischen eigentlich obsoleten Fortran77-Standard ist die Aufteilung einer Programmzeile genau festgelegt:

Spalten 1–5: Zeilennummern (optional; z.B. als Zielmarken für `goto`-Sprünge)

Spalte 6: Fortsetzungszeichen (wenn nötig)

Spalten 7–72: eigentlicher Befehlstext

Wenn ein Befehl für eine Zeile zu lang wird, kann er in einer (oder mehreren) direkt anschließenden Fortsetzungszeilen fortgesetzt werden. Bei solchen Fortsetzungszeilen (und nur bei diesen) steht in Spalte 6 ein beliebiges Zeichen (verbreitet sind: ein unterschiedsloses „x“, oder eine Durchnumerierung mit „1“, „2“, usw. bei mehreren Fortsetzungszeilen).

Kommentare, deren Inhalt vom compiler komplett ignoriert wird, stehen beim festen Zeilenformat in speziellen Kommentarzeilen. Bei solchen Kommentarzeilen steht in der ersten Spalte der Buchstabe „c“.

3.2.2 freies Format

Ab Fortran90/95 wurde ein freieres Zeilenformat eingeführt. Dessen wesentliche Merkmale sind u.a.:

- die Zeilen sind länger (132 Spalten),
- innerhalb der Zeile kann der Programmtext irgendwo stehen (von Spalte 1 bis Spalte 132),
- Kommentare werden nicht durch „c“, sondern durch ein Ausrufungszeichen „!“ eingeleitet und müssen nicht in eigenen Zeilen stehen (z.B. kann so an eine ausführbare Programmzeile noch ein kleiner Kommentar angehängt werden).

- Fortsetzungszeilen werden anders gekennzeichnet: eine nicht fertige Zeile endet mit einem Schrägstrich „/“ und die folgende(n) Fortsetzungszeile(n) beginnt/-en mit einem Schrägstrich.

Manche compiler sind in der Lage, festes und freies Format automatisch zu erkennen und korrekt zu verarbeiten. Andere brauchen dafür z.B. geeignete Compileroptionen. In jedem Fall sollte man festes und freies Format nicht innerhalb eines Programms vermischen.

4 Konstante und Variable

4.1 Grundkonzepte

Ein- und Ausgabedaten sowie (schon bei einfachsten Rechnungen) Zwischenergebnisse müssen beim Programmlauf temporär gespeichert werden. Diese Daten stehen dann tatsächlich (zusammen mit dem ausführbaren Programm) im Speicher des Rechners. Als Programmierer (einer Hochsprache) müssen Sie sich aber nicht um die Organisation dieser Abspeicherung konkret kümmern. Stattdessen speichern Sie alle Ihre Daten dadurch ab, daß Sie sie geeigneten *Variablen* zuweisen.

Sie erzeugen eine neue Variable dadurch, daß Sie ihr einen Namen geben und einen Variablentyp (aus einer sehr geringen Anzahl vorgegebener Möglichkeiten) wählen. Dadurch ist der compiler in der Lage, für diese Variable einen Speicherbereich passender Größe zu reservieren. Die in dieser Variable aktuell abgelegten Daten können Sie im weiteren Programmverlauf durch Angabe des Variablennamens beliebig oft wieder erhalten.

Jede Variable sollte bei ihrem ersten Gebrauch einen Wert zugewiesen bekommen (*initialisiert* werden). In vielen compilern ist es „legal“, uninitialisierte Variable in arithmetischen Ausdrücken (z.B. auf der rechten Seite eines Gleichheitszeichens in einer Rechnung) zu verwenden; andere compiler geben in solchen Situationen eine Warnung aus (zumindest wenn sie per compiler-Option dazu aufgefordert werden). Manche compiler (nicht notwendigerweise dieselben und keineswegs alle) initialisieren jede neue Variable bei ihrem ersten Erscheinen automatisch auf den Wert Null. Daher sollte der Programmierer immer selbst darauf achten, in jedem Programmteil jede Variable in gewünschter Weise zu initialisieren.

Feste Zahlenwerte und feste Texte (*Konstanten*) können in ausführbaren Anweisungen an allen Stellen stehen, an denen auch Variable verwendbar sind (und umgekehrt). Genau wie Variable haben Konstanten auch einen Typ, der jedoch nicht vereinbart wird, sondern aus der Schreibweise der Konstanten hervorgeht. Dies wird unten bei den entsprechenden Variablentypen erläutert.

4.2 Variablennamen

Variablennamen können maximal 31 Zeichen lang sein. Das erste Zeichen muß ein Buchstabe sein, alle anderen können Buchstaben, Zahlen oder der Unterstrich „_“ sein. Da Sie zusätzlich zu einfachen Variablen auch ein- und mehrdimensionale Felder beliebiger Größe vereinbaren können, ist die Menge von Daten, die Sie speichern können, allein von der Größe Ihres Computerspeichers bestimmt.

Es ist dringend zu empfehlen, Variablennamen nach ihrer Funktion im Programm sinnvoll zu wählen, auch wenn sich dadurch die Schreibarbeit erhöht.

4.3 Variablentypen und -deklaration

Mit einer oder mehreren Zeilen folgender Syntax

Variablentyp Variablenname,Variablenname,...

können in der Deklarationssektion ein oder mehrere beliebige Variablennamen vereinbart und gleichzeitig folgenden möglichen Variablentypen zugewiesen werden:

integer: ganze Zahlen

real: (oder real(kind=4)) Zahl mit ca. 7 signifikanten Nachkommastellen (single precision, belegt 4 Bytes im Speicher, s.u.)

real(kind=8): Zahl mit ca. 15 signifikanten Nachkommastellen (double precision, belegt 8 Bytes im Speicher, s.u.)

complex: komplexe Zahl (zusammengesetzt aus einer real-Zahl als Realteil und einer real-Zahl als Imaginärteil, wird hier nicht behandelt)

character(len=n): Textvariable mit n Zeichen Länge

logical: logische Variable (werden hier nicht behandelt)

(*Vorsicht:* Im alten Fortran77-Standard war die Syntax speziell bei der Deklaration von real-Variablen etwas anders: real∗4 und real∗8, bzw. `real` und `double precision`. Das ist mit den meisten Fortran90/95-compilern auch noch erlaubt.)

4.3.1 Integervariable

Eine Integervariable kann positive oder negative ganze Zahlen aufnehmen. Je nach compiler werden dafür unterschiedlich große Bereiche im Speicher zugewiesen, in jedem Fall aber eine begrenzte Anzahl von Bytes. Daher kann der Absolutwert einer solchen ganzen Zahl nicht beliebig groß sein; für übliche Anwendungen ist der Bereich jedoch ausreichend.

Eine Integerkonstante ist schlicht eine Ziffernfolge ohne Dezimalpunkt, mit oder ohne Vorzeichen.

4.3.2 real-Variable

Bei real-Variablen hat man die Wahl zwischen wenigstens zwei Genauigkeitsstufen, meist als single und double precision bezeichnet. Typischerweise können diese beiden Möglichkeiten mit den Vereinbarungen `real(kind=4)` bzw. `real(kind=8)` ausgewählt werden (oft sind `real(kind=single)` bzw. `real(kind=double)` synonym dazu). Typischerweise reserviert der Compiler dafür 4 bzw. 8 Bytes im Speicher, was den Darstellungsbereich und die Genauigkeit erheblich beeinflußt. Typische Werte sind (schwankt je nach compiler-Implementation, also nur grobe Richtwerte):

	4 bytes	8 bytes
kleinste positive Zahl	1.2×10^{-38}	2.2×10^{-308}
größte positive Zahl	$3.4 \times 10^{+38}$	$1.8 \times 10^{+308}$
relative Genauigkeit	1.2×10^{-7}	1.1×10^{-16}

Dummerweise legt der Fortran90/95-Standard keinerlei Implementationsdetails für die Genauigkeit von `real`-Variablen fest. Daher kann die tatsächliche Realisierung auch ganz anders aussehen, sowohl hinsichtlich der Anzahl der Bytes als auch der `kind`-Bezeichnungen (typischerweise tut sie das z.B. auf Cray-Supercomputern und bei NAG-Compilern).

Anfängern ist daher zu empfehlen, Variablen für Kommazahlen einfach als `real` zu vereinbaren (ohne weitere Verzierungen) und ansonsten unbedingt das jeweils vorliegende compiler-Manual in diesem Punkt zu konsultieren.

Eine real-Konstante, z.B. der Zahlenwert 1.2×10^{-38}, kann je nach gewünschter Genauigkeit ebenfalls unterschiedlich notiert werden:

single precision: 1.2e-38

double precision: 1.2d-38

4.3.3 Textvariable

Textvariable können neben Buchstaben auch Zahlen und Sonderzeichen aufnehmen (z.B. Ausrufezeichen usw.). Je nach compiler können bestimmte Sonderzeichen jedoch auch Formatierungsaufgaben erfüllen. Bei der Vereinbarung einer Textvariablen ist immer deren Länge mit anzugeben, in einer der folgenden Formen:

```
character::bla,laber
character(len=42)::lall,ruelps
```

Die erste dieser Zeilen vereinbart zwei Textvariablen von jeweils 1 Zeichen Länge. Die zweite Zeile vereinbart zwei je 42 Zeichen lange Variablen. Die Zuweisung von zu langen Texten an Textvariable führt bestenfalls zum Abschneiden des Textes, schlimmstenfalls zu unvorhersagbaren Resultaten. Die Zuweisung von zu kurzen Texten ist dagegen völlig unschädlich (der Rest wird mit Leerzeichen gefüllt).

Jeder durch Anführungszeichen eingeschlossene Text ist für Fortran eine Textkonstante, jeder Text ohne Anführungszeichen entweder ein Variablenname oder eine Anweisung. Also hat die Zuweisung eines Werts zu einer Textvariablen folgendermaßen auszusehen:

```
...
character(len=10)::sinn
...
sinn = 'Unsinn!'
...
```

4.4 implizite Typvereinbarung

Auch im Fortran90/95-Standard ist es noch möglich, Variablen ohne Deklaration zu verwenden. Dann nimmt der compiler an, daß Variablen, deren Namen mit den Buchstaben `i` bis `n` beginnen, `integer` sind und alle anderen `real`. Mit dem `implicit`-statement ist es möglich, diese implizite Typvereinbarung zu ändern oder zu ergänzen, dies soll hier aber nicht weiter erklärt werden.

Das scheint auf den ersten Blick eine sinnvolle Erleichterung zu sein und wurde deshalb in älteren Fortran-Programmen auch extensiv benutzt. Heute hat sich jedoch die Ansicht

durchgesetzt, daß diese implizite Typvereinbarung beim Schreiben von Programmen doch eher schadet als nützt. In völliger Abwesenheit impliziter Typvereinbarungen müssen alle Variablen eines Programmabschnitts im Deklarationsteil dieses Abschnitts explizit vereinbart werden. Das ist auptsächlich aus zwei Gründen sehr vorteilhaft:

- Die häufigsten Programmierfehler sind simple Schreibfehler. Tippfehler bei Variablennamen und auch bei Fortran-statement-Namen kann der compiler genau dann einigermaßen zuverlässig erkennen, wenn er *nicht* annimmt, daß jeder neu auftauchende Name eine Variable von implizit vereinbartem Typ ist.
- Einer der häufigsten Fehler beim Zusammensetzen neuer Programme unter Verwendung bereits vorhandener Programmteile (z.B. subroutines) ist es, in einen vorhandenen Programmteil eine neue Variable einzuführen und mit Werten zu belegen, deren Name aber bereits in diesem Programmteil existiert und für gänzlich andere Zwecke reserviert ist. Mit einer vollständigen Variablenliste in der Deklarationssektion kann dieser Fehler viel leichter vermieden werden.

Jegliche implizite Typvereinbarung in einem Programmabschnitt kann durch den nichtausführbaren Befehl `implicit none` abgeschaltet werden. Es ist daher dringend empfehlenswert, diese Zeile in die Deklarationssektion jedes Programmabschnitts aufzunehmen.

4.5 Typumwandlung

4.5.1 automatisch

Fortran wandelt in einigen Situationen automatisch den Typ der Zahlendarstellung *intern* um, bevor bestimmte Operationen ausgeführt werden. Oft hat das für den Programmierer keine Konsequenzen, manchmal kann es jedoch auch fatal sein. Eine beliebte Falle ist die automatische Umwandlung von real-Werten in integer-Werte, insbesondere weil sie nicht durch Rundung, sondern durch Abschneiden der Nachkommastellen erfolgt. Nach Abarbeitung dieses Programmsegments

```
...
integer i
...
i = 0.9d0
...
```

enthält daher die Variable `i` nicht den Wert 1, sondern den Wert 0 (die Schreibweise `0.9d0` mit dem eigentlich überflüssigen Exponenten ist ein Standardtrick, um sicherzustellen, daß Fortran die Zahlenkonstante 0.9 als double precision behandelt, was in anderen Situationen als dieser durchaus sinnvoll sein kann.)

4.5.2 explizit

Mit einigen eingebauten Fortran-Funktionen kann der Programmierer die Typumwandlung explizit kontrollieren (unvollständige Beispiel-Aufzählung):

real(i): verwandelt den ganzzahligen Wert in der Variablen i in eine real-Zahl ;

int(x): konstruiert aus der real-Zahl in x einen ganzzahligen Wert, durch Abschneiden der Nachkommastellen;

nint(x): rundet die real-Zahl in x auf die nächste ganze Zahl, so wie man es eigentlich erwarten würde.

4.6 Felder

Anstatt nur einen Wert in einer Variablen abzuspeichern, kann man in geeignet vereinbarten Variablen auch mehrere Werte abspeichern; solche Variablen nennt man (ohne strenge Unterscheidungen) Felder, Vektoren, Matrizen, arrays, ...

Solche Felder kann man z.B. folgendermaßen deklarieren:

```
integer,dimension(3)::i
integer,dimension(10,5)::j
real,dimension(2,3,2)::a
```

In der ersten Zeile wird eine eindimensionale integer-Feldvariable `i` mit maximal 3 Einträgen vereinbart, in der zweiten eine zweidimensionale integer-Feldvariable `j`, die man sich als 10×5-Matrix vorstellen kann. In der dritten Zeile wird eine dreidimensionale real-Feldvariable `a` deklariert, die insgesamt maximal 12 Einträge aufnehmen kann.

Im nachfolgenden Programm kann man auf die einzelnen Werte in diesen Feldvariablen dadurch zugreifen, daß man den Feldvariablennamen zusammen mit geeigneten Zahlen für die *Index*-Werte angibt, also z.B. `a(2,3,1)`.

Der unglaublich große Nutzwert dieser unscheinbaren Maßnahme liegt darin, daß man anstelle fester Konstanter für die Indexwerte auch (integer-)Variablen verwenden darf. Dadurch wird es möglich, z.B. in Schleifen (siehe Kontrollstrukturen) dieselben Rechenoperationen auf eine große Anzahl von Zahlen anzuwenden (die dann in solchen Feldern gespeichert sind) – ohne daß man jeden einzelnen dieser Schritte wiederholt hinschreiben muß, ja sogar ohne daß man vorher wissen muß, wieviele Zahlen man im konkreten Anwendungsfall dann tatsächlich bearbeiten wird.

(Wie die Nomenklatur schon andeutet, sind diese Vektor- bzw. Matrix-Konstrukte und ihre Indices ganz analog zu ihren Namensvettern in der Mathematik zu verstehen. Das erleichtert es enorm, z.B. mathematische Summenausdrücke wie $\sum_i x_i$ direkt in Fortran-Progammabschnitte zu übersetzen.)

5 Rechenoperationen

5.1 Grundrechenarten mit Zahlen

Addition, Subtraktion, Multiplikation und Division werden durch die Zeichen +, -, ∗, / zwischen Zahlenvariablen oder Konstanten ausgeführt.

In einigen Situationen ist auch hier der Variablen-/Konstanten-Typ zu berücksichtigen. Insbesondere ist die Operation `i/j` mit zwei integer-Variablen `i` und `j` eine sogenannte integer-Division, die etwas anders funktioniert als erwartet. Das in der Mathematik

sonst übliche Resultat kann man in dieser Situation nur durch explizite Typumwandlung erzwingen: `real(i)/real(j)`.

Die Potenzfunktion x^y, die man eigentlich auch als Funktion im Sinne des nächsten Teilabschnitts auffassen könnte, wird jedoch wie die Grundrechenarten auch durch ein einfaches Verknüpfungszeichen realisiert: `x**y` (wobei x und y integer- oder real-Variablen sein können).

Arithmetische Ausdrücke werden in der folgenden Reihenfolge ausgewertet:

1. Der Inhalt aller Ausdrücke in Klammern wird berechnet, von der innersten Klammer nach außen (d.h.: vorhandene Klammern haben die gewünschte Prioritätswirkung);
2. alle Potenzierungen werden berechnet, von rechts nach links;
3. alle Multiplikationen und Divisionen werden ausgewertet, von links nach rechts;
4. alle Additionen und Subtraktionen werden berechnet, von links nach rechts.

5.2 Funktionen

Fortran enthält zahlreiche Ausdrücke der üblichen Mathematik als fest eingebaute Funktionen (im Fortran-Sinn, also als vorgegebene Funktions-Subroutinen, die eigentlich nach den dafür geltenden Regeln funktionieren, siehe Kapitel „Subroutinen und Funktionen“; das ist aber für uns fast ohne Belang).

Alle diese Funktionen werden aufgerufen durch eine Zeichenfolge der Form „`funk(arg,arg,...)`“, wobei `funk` der Name der Funktion ist und in Klammern ein oder mehrere Variablen oder Konstanten als input-Argumente an diese Funktion übergeben werden.

An dieser Stelle sind im Prinzip wieder die Variablentypen wichtig: Eine Fortran-Funktion akzeptiert eigentlich nur Variablen eines bestimmten Typs und gibt als Resultat auch wieder nur einen Wert eines bestimmten Typs zurück. Dementsprechend gibt es tatsächlich viele derartig spezialisierte Funktionen, z.B.: `dabs(x)` erzeugt den double-precision-Absolutwert einer double-precision-Variablen `x`, `iabs(i)` erzeugt den integer-Absolutwert einer integer-Variablen `i`, usw.

Zum Glück gibt es jedoch auch allgemeine (*generische*) Varianten dieser Funktionen, im obigen Beispiel wäre dies: `abs(a)`. Dieser Aufruf funktioniert in genau dieser Form für verschiedene Variablentypen: Ist die Variable `a` ganzzahlig, gibt `abs(a)` auch ein ganzzahliges Resultat zurück. Ist `a` eine real-Variable, ist auch das Ergebnis von `abs(a)` vom real-Typ, usw.

Die folgende Liste derartiger Funktionen ist nicht vollständig, sondern enthält nur die für unsere Zwecke wichtigsten Funktionen:

abs(a): Absolutwert (Betrag)

acos(a): Arcus cosinus (Umkehrfunktion des Cosinus)

asin(a): Arcus sinus (Umkehrfunktion des Sinus)

atan(a): Arcus tangens (Umkehrfunktion des Tangens)

cos(a): Cosinus

exp(a): Exponentialfunktion („e hoch a“)

log(a): natürlicher(!) Logarithmus (Basis e)

log10(a): dekadischer Logarithmus (Basis 10)

max(a1,a2,...): gröster Wert aus den Zahlen a1, a2, ...

min(a1,a2,...): kleinster Wert aus den Zahlen a1, a2, ...

sign(a,b): a mit dem Vorzeichen von b

sin(a): Sinus

sinh(a): Sinus hyperbolicus

sqrt(a): Quadratwurzel

tan(a): Tangens

tanh(a): Tangens hyperbolicus

6 Eingabe, Ausgabe und files

6.1 I/O-units

Ein- und Ausgabe von Daten erfolgt in Fortran über files (Dateien). Im Betriebssystem haben aus Benutzersicht alle files einen lesbaren Namen. Innerhalb eines Fortran-Programms werden files jedoch über sogenannte *unit numbers* angesprochen, die (positive) ganze Zahlen sind

Bei vielen (aber nicht allen) compilern ist ohne anderslautende Vereinbarungen die unit number 5 für Eingabe über die Tastatur und die unit number 6 für Ausgabe auf den Bildschirm voreingestellt. Da dies jedoch nur fast, aber nicht ganz universell gilt, ist es empfehlenswert, anstelle von 5 bzw. 6 einen Stern „$*$“ anstelle der unit number zu verwenden. Damit ist dann auf jeden Fall die Tastatur bzw. der Bildschirm verknüpft.

Daher empfiehlt es sich aber auch, die unit numbers 5 und 6 nicht für eigene Zwecke zu verwenden; das wäre zwar möglich, führt aber natürlich zu Verwirrung. Natürlich ist es auch unsinnig, eine unit number mit zwei verschiedenen file-Namen zu verbinden bzw. umgekehrt. Ansonsten können die unit numbers jedoch nach Belieben gewählt werden (eine frühere Beschränkung darauf, daß die unit numbers nur Zahlen kleiner als 100 sein durften, existiert nicht mehr).

6.2 Öffnen von files

Natürlich muß dann wenigstens einmal im Programmablauf (vor Verwendung des jeweiligen files) eine Verknüpfung zwischen dem file-Namen und einer unit number hergestellt werden. Dies geschieht mit dem `open`-statement:

```
open(unit=2,file='blabla.txt',status='old')
```

In diesem Beispiel wird das file mit dem (Betriebssystem-)Namen „blabla.txt“ mit der Fortran-unit-number 2 verbunden. Wie der Befehlsname `open` suggeriert, ist ab diesem Befehl im gesamten weiteren logischen Verlauf des Programms (also global, über das Hauptprogramm und alle Unterprogramme hinweg) diese unit number „offen“ für Ein- und Ausgabe, die dann in das angegebene file gelenkt wird.

Der file-Name ist in diesem Beispiel eine Textkonstante. An deren Stelle könnte aber auch eine Textvariable stehen, was die Flexibilität erhöht: Dadurch kann z.B. das Programm interaktiv vom Benutzer Namen für zu öffnende files erfragen.

Der optionalen zusätzlichen `status`-Angabe können (u.a.) folgende Werte zugewiesen werden, die zu unterschiedlichem Verhalten der open-Anweisung und des ganzen Programms führen:

new: Mit der open-Anweisung wird ein neues file eröffnet. Gibt es bereits ein file gleichen Namens (im aktuellen Verzeichnis, wenn keine explizite Pfad-Information im Dateinamen enthalten ist), bricht das Programm mit einer entsprechenden Fehlermeldung ab. Dadurch kann man unbeabsichtigtes Überschreiben vorhandener Daten verhindern.

old: Mit der open-Anweisung wird ein bereits vorhandenes file mit dem angegebenen Namen geöffnet. Existiert ein solches file nicht, bricht das Programm mit einer entsprechenden Fehlermeldung ab. Das ist besonders sinnvoll bei files, von denen das Programm zu Beginn essentielle input-Daten einlesen soll.

unknown: Wenn das file noch nicht existiert, wird es angelegt. Existiert es schon, wird dieses vorhandene file verwendet; damit wird sein bereits vorhandener Inhalt überschrieben, wenn man danach Ausgabe auf dieses file lenkt.

Wenn man nicht ganz genau weiß, was man tut, empfiehlt es sich, ein file entweder nur für Eingabe oder aber nur für Ausgabe zu öffnen (in Fortran90/95 gibt es dafür bestimmte zusätzliche Optionsangaben, die hier aber nicht besprochen werden sollen; es reicht, wenn man diese Vereinbarung mit sich selber trifft).

Dann ist es in der Regel sinnvoll, Eingabefiles immer mit `old` zu öffnen. Erzeugt man mit `open` ein neues file, ist es zunächst leer. Eingabe von einem leeren file führt dann zwangsläufig zu einem Abbruch mit Fehlermeldung, die aber (je nach compiler) meist schwerer verständlich ist als die Fehlermeldung, die man erhält, wenn das Programm ein mit `old` zu öffnendes file nicht findet.

Ganz analog sollte man dann Ausgabefiles in der Regel mit `unknown` öffnen, es sei denn, man möchte bei einem erneuten Programmlauf seine alten output-files keinesfalls überschreiben. Dann muß man diese files aber natürlich umbenennen oder in ein anderes Verzeichnis bewegen.

6.3 Schließen von files

Mit dem `close`-Befehl kann man offene Dateien bzw. units wieder schließen und damit auch die jeweilige Assoziation zwischen file-Name und unit number wieder aufheben:

```
...
open(unit=42,file='input.dat',status='new')
...
close(42)
...
```

Bereits sofort danach könnte, wenn nötig, unit number 42 mit einem anderen file-Namen und/oder der file-Name „input.dat“ mit einer anderen unit number verbunden werden.

Beim `end`-Befehl am Hauptprogrammende (*nicht* am Ende von subroutines oder functions!) werden alle noch offenen files automatisch geschlossen. Trotzdem sollte man sich angewöhnen, das mit einem explizitien `close`-Befehl selber zu tun.

Man sollte files generell auch nicht länger als nötig offen halten, zumindest bei Programmen, die Laufzeiten von mehr als einigen Minuten haben: Auf wieder geschlossene files können andere Programme noch zu Laufzeit des ersten Programms zugreifen. Bei noch offenen files kann das zu unvorhersagbaren Resultaten führen.

6.4 Eingabe und Ausgabe

Die eigentliche Ein- und Ausgabe erfolgt dann mit den Befehlen `read` bzw. `write`, die eine praktisch identische Syntax haben:

```
...
open(unit=137,file='eingabe.in',status='old')
...
open(unit=248,file='ausgabe.out',status='unknown')
...
write(248,*)'nur ein dummes Beispiel'
...
read(137,*)variable1,variable2
...
```

Wie dieses Beispiel suggeriert, ist das erste Argument in Klammern nach dem read- und write-Befehl die gewünschte unit number. Das zweite Argument in Klammern sollte eigentlich eine sogenannte Formatangabe sein, mit der man z.B. festlegen kann, mit wievielen Dezimalstellen eine Zahl ausgegeben werden soll, in welchen Spalten der Ausgabezeile sie genau zu stehen hat, usw. Dieses etwas längliche Thema wird hier komplett weggelassen. Man kann als Alternative die Formatierung komplett dem Computer überlassen; dafür schreibt man dann einen Stern „*“ als Formatangabe. Eine unformatierte Ausgabe auf den Bildschirm erfolgt daher mit `write(*,*)`.

In der Regel entspricht ein `read`- bzw. `write`-Befehl genau einer Ein- bzw. Ausgabezeile. Ein `write(*,*)a,b` schreibt daher die momentan in den Variablen `a` und `b` gespeicherten Zahlen nacheinander in eine einzige Ausgabezeile. Je ein write-Befehl für `a` und `b` würde zwei Ausgabezeilen mit je einer Zahl erzeugen. Die Verhältnisse liegen beim read-Befehl eigentlich ähnlich. Gerade bei einer unformatierten Eingabe wie im obigen Beispiel ist es jedoch bei vielen compilern erlaubt, daß ein read-statement für mehrere Variablen diese nicht nur aus einer, sondern auch aus mehreren Eingabezeilen lesen kann (wenn in diesen Zeilen jeweils zuwenig Zahlen stehen).

7 Kontrollstrukturen

Die bis hierher angenommene, einfache sequentielle Abarbeitung eines Programmtextes ist häufig nicht ausreichend, um die gewünschten Algorithmen zu realisieren. Jede Programmiersprache stellt daher mindestens einige Standardkonstrukte zur Verfügung, die den

Abarbeitungsablauf des Programms beeinflussen und in den meisten modernen Sprachen sehr ähnlich aussehen.

7.1 Bedingte Verzweigungen

7.1.1 allgemeine Struktur eines `if`-Blocks

Häufig soll ein Programm unterschiedliche Dinge tun, je nachdem, welchen Wert bestimmte Zwischenergebnisse oder Steuerungsvariablen haben. Für solche Fälle gibt es bedingte Verzweigungen, die in Fortran folgendermaßen aussehen:

```
if (Bedingung1) then
  Teil1
else if (Bedingung2) then
  Teil2
else
  TeilX
end if
```

Hier wird zunächst Bedingung1 überprüft; trifft sie zu, wird der hier mit Teil1 bezeichnete Programmteil (aus beliebig vielen Anweisungszeilen) ausgeführt; danach werden die nach dem `end if`-statement stehenden weiter bearbeitet. Trifft Bedingung1 nicht zu, wird zum nächsten `else if` gesprungen und die dort stehende Bedingung2 überprüft; ist sie wahr, wird Teil2 ausgeführt und dann nach `end if` weitergemacht, usw. Man beachte, daß Bedingung2 nur dann überprüft wird, wenn Bedingung1 *nicht* schon zutreffend war.

Innerhalb dieses `if/end if`-Konstrukts kann es eine beliebige Anzahl von `else if`-statements geben. Es kann auch gar keine `else if`-Zeile geben. In jedem Fall wird der hinter dem `else`-statement stehende TeilX genau dann (bedingungslos) ausgeführt, wenn *keine* der Bedingungen zutrifft, die bei einem der `if`- oder `else if`-statements stehen. Dieser `else`-Abschnitt seinerseits kann jedoch ebenfalls fehlen; in diesem Fall wird innerhalb des ganzen if-Blocks gar nichts getan, wenn keine der Bedingungen zutreffen sollte.

Für die eigene Übersicht empfiehlt es sich, die einzelnen Befehle innerhalb des if-Blocks um einen festen Betrag nach rechts einzurücken, die Kontrollzeilen mit if/else/endif jedoch nicht, wie im Beispiel angedeutet. Natürlich können in jedem dieser Teilabschnitte weitere Kontrollstrukturen vorkommen, insbesondere auch weitere if-Blöcke. Spätestens dann ist eine konsequente, gestaffelte Einrückung unerläßlich.

Soll beim Zutreffen einer einzigen Bedingung nur ein einziges statement abgearbeitet werden und ansonsten nichts geschehen, gibt es eine auf eine einzige Zeile verkürzte Form des if-Blocks:

```
if (Bedingung) statement
```

7.1.2 Logische Operatoren und Bedingungen

Strenggenommen ist eine Bedingung ein logischer Ausdruck, der nach einem bestimmten Satz von Regeln durchaus verschiedene Formen annehmen kann. Hier reicht es jedoch, sich

vorzustellen, daß eine Bedingung einfach ein Vergleich zwischen zwei Variablen oder zwischen einer Variablen und einer Konstanten ist. Dabei sind verschiedene Vergleichsoperatoren erlaubt, bei denen sich allerdings die (bevorzugte) Syntax von Fortran77 auf Fortran90/95 geändert hat (die alten Operatornamen setzen sich aus den Anfangsbuchstaben der englischen Bezeichnungen less than, equal, greater than, not, usw. zusammen):

Bedingung zutreffend (wahr), wenn	alt	neu
`a` kleiner als `b`	`a.lt.b`	`a < b`
`a` kleiner oder gleich `b`	`a.le.b`	`a < = b`
`a` größer als `b`	`a.gt.b`	`a > b`
`a` größer oder gleich `b`	`a.ge.b`	`a > = b`
`a` gleich `b`	`a.eq.b`	`a == b`
`a` nicht gleich `b`	`a.ne.b`	`a /= b`

Bei solchen Vergleichen sollte man einige beliebte Fehler vermeiden, z.B. ist es wegen der begrenzten Computergenauigkeit gefährlich, direkt abzufragen, ob eine berechnete reelle Größe gleich einer vorgegebenen Konstanten ist – das wird in der Praxis eher selten der Fall sein, auch dann, wenn dieses Ergebnis theoretisch exakt vorliegen sollte (bei integer-Resultaten und sich daraus ergebenden integer-Vergleichen gibt es diese Gefahr natürlich (fast) gar nicht). In solchen Fällen ist es besser, abzufragen, ob die Differenz zwischen dem berechneten Resultat und der vorgegebenen Konstanten kleiner ist als eine (von der Computergenauigkeit bestimmte) sehr kleine Zahl. Da man von vornherein nicht weiß, ob das berechnete Resultat aufgrund quasi-zufälliger Rundungsfehler etwas zu klein oder etwas zu groß ausfallen wird, ist es dabei dringend anzuraten, vor dem Vergleich den Absolutbetrag der Differenz zu bilden.

7.2 Schleifen

In vielen Programmsituationen sollen gleiche oder sehr ähnliche Dinge wiederholt getan werden, z.B. schlicht bei der Bearbeitung großer Zahlenmengen (in Feldern), aber auch bei Iterationen oder Rekursionen, usw. Dies könnte man durch entsprechend häufige, explizite Wiederholungen desselben (oder jedesmal nur leicht variierten) Programmsegments erreichen. Das hat jedoch mindestens zwei erhebliche Nachteile: (1.) Das Programm kann dadurch sehr schnell extrem lang und entsprechend unübersichtlich werden. (2.) Sehr oft weiß man die Anzahl der nötigen Wiederholungen vorher noch gar nicht, oder man will das Programm so allgemein halten, daß es unverändert beliebige Anzahlen von Wiederholungen zuläßt. Für diese Zwecke gibt es in allen Sprachen (Wiederholungs-)*Schleifen*.

Alle Schleifen haben in Fortran folgende generelle Form:

```
do
   TeilX
end do
```

Dabei werden die Befehle in TeilX nacheinander abgearbeitet. Bei Erreichen des `enddo`-statements springt das Programm wieder zum `do`-statement zurück und das Ganze wiederholt sich. In dieser Form selber ist die Schleife allerdings sinnlos, da dann die in TeilX stehenden Befehle unendlich oft wiederholt würden. Mit leichten Modifikationen lassen sich daraus jedoch sehr sinnvolle Formen machen (die in anderen Sprachen z.T. scheinbar unterschiedliche Konstrukte sind). Die beiden wichtigsten Formen sollen hier vorgestellt werden.

7.2.1 do-while/until-Schleifen

Das mindeste, was man sinnvollerweise tun muß, ist, die ansonsten nie endenden Wiederholungen des Schleifeninhalts TeilX irgendwann abzubrechen, woraufhin das Programm nach dem `enddo`-statement fortgesetzt wird. Genau dies bewirkt der `exit`-Befehl (ein statement für sich, ohne weitere Optionen o.ä.).

Natürlich möchte man aber i.A. die Schleife nicht gleich beim ersten Durchlauf auf diese Weise beenden. Das kann man dadurch erreichen, daß man vor das `exit` ein if-statement (also eine Bedingungsabfrage) einschiebt. Damit haben wir jetzt folgende Konstruktion:

```
do
   Teil1
   if (Bedingung) exit
   Teil2
end do
```

Logischerweise sollte sich bei den Wiederholungen des Schleifenkörpers irgendwann einmal die Bedingung von falsch auf wahr ändern, sonst hat man immer noch entweder eine Endlosschleife oder eine Schleife mit nur einem (unvollständigen) Durchlauf. Sinnvolle Bedingungen sind daher z.B. Konvergenzabfragen in iterativen Algorithmen. Ein schlichtes Abzählen der Wiederholungen und Beendigung der Wiederholungen nach einer gewissen, vorgegebenen Anzahl erreicht man dagegen einfacher mit einer echten Zählschleife, s.u.

Daraus ergibt sich auch, daß eine solche Schleife immer die Gefahr in sich birgt, zu einer Endlosschleife zu entarten, nämlich dann, wenn die Bedingung beim `exit`-Befehl nie zutrifft, was durch unvorgesehene numerische Probleme oder ganz banale Programmierfehler leicht passieren kann. Selbst dann, wenn die Bedingung irgendwann einmal tatsächlich zutreffen wird, kann die Anzahl der Schleifendurchläufe und damit die Programmausführungsdauer u.U. erheblich größer werden als geplant. Daher sollte man bei solchen Schleifen immer sehr vorsichtig sein.

Ist die if-exit-Zeile die allererste innerhalb der Schleife (Länge von Teil1 ist Null), wird die Abbruchbedingung gleich zu Beginn geprüft. Diese Situation wird *while*-Schleife genannt. Tatsächlich gibt es in Fortran90/95 dafür eine spezielle Syntax, in dem das Wort `while` explizit vorkommt (dasselbe gilt für viele andere Sprachen). In der hier vorgestellten Form ist die while-Schleife jedoch einfacher und gleichzeitig flexibler.

Ist die if-exit-Zeile die letzte Zeile innerhalb der Schleife, wird die Abbruchbedingung als letztes geprüft. Dies ist eine sogenannte *until*-Schleife (andere Sprachen haben dafür häufig auch eine eigene Syntax).

Gelegentlich nützlich ist auch der Befehl `cycle` (sinnvollerweise mit einer if-Abfrage zu kombinieren, wie der `exit`-Befehl). Er bewirkt kein Verlassen der Schleife, sondern einen Rücksprung an den Schleifenanfang.

7.2.2 Zählschleifen

Steht von vornherein fest, daß ein gewisser Programmteil n mal wiederholt werden soll (oder wenn man die Anzahl von Wiederholungen einer Schleife auf eine vorgegebene Maximalzahl begrenzen will, um Endlosschleifen zu vermeiden), bietet sich eine Schleife mit eingebautem Wiederholungszähler an:

```
do i=n1,n2
  TeilX
end do
```

Beim ersten Durchlauf erhält die integer-Variable `i` (die natürlich auch einen ganz anderen Namen haben kann) den in der integer-Variablen `n1` gespeicherten Wert. Beim zweiten Durchlauf hat `i` den Wert $n1 + 1$, usw. Beim letzten Durchlauf hat `i` den Wert $n2$. Danach wird das Programm nach dem `enddo`-statement fortgesetzt.

An den Stellen der Variablen `n1` und/oder `n2` können natürlich auch integer-Konstanten stehen. Eine Schleife, die mit `i=1,n` beginnt, wird genau n-mal ausgeführt. Eine Schleife, bei der $n1 = n2$ ist, wird genau einmal ausgeführt. Eine Schleife, bei der $n1 > n2$ ist, wird keinmal ausgeführt.

Die statements `exit` und `cycle` können auch in dieser Art von Schleifen verwendet werden, mit analogen Effekten. `cycle` kann hier z.B. nützlich sein, um die Schleifenausführung für gewisse Werte der Zählvariablen (hier `i`) teilweise oder ganz zu überspringen.

(Im Prinzip ist die Schleifensyntax flexibler als hier vorgestellt. Z.B. kann man mit einer weiteren Zahlenangabe nach n1 und n2 das Inkrement, um das i bei jedem Durchlauf erhöht wird, von +1 auf andere Werte ändern, sogar auf negative. Das ist nicht unbedingt nötig, kann aber als verkürzende Schreibweise manchmal nützlich sein. – Einige compiler erlauben nach wie vor die Unart, als Zählvariable real-Werte zu verwenden, was man aber komplett vermeiden sollte; in zukünftigen Fortran-Standards wird es verboten sein.)

7.3 Sprünge (zu vermeiden!)

In antiken Fortran-Standards gab es fast keine Kontrollstrukturen. Stattdessen wurden sie durch Sprungbefehle (von denen es diverse verschiedene Typen gab) simuliert. Das hat sich in der Praxis als für den menschlichen Leser sehr unübersichtlich und für den compiler kaum zu analysieren erwiesen. Daher versuchen neuere Fortran-Standards, von Sprungbefehlen gänzlich abzusehen.

Dennoch hat zumindest das einfache `go to` bis in den aktuellen Fortran90/95-Standard überlebt, wenn auch nur als ungern gesehene Notlösung:

```
    ...
    go to 100
    write(*,*)'Hilfe'
100 e=m*c**2
```

In diesem Beispielfragment hat die letzte Zeile die Zeilennummer 100 erhalten. Sie wird im `goto`-statement in der ersten Zeile verwendet, um von dieser ersten Zeile direkt in die letzte Zeile zu springen. Die mittlere Zeile wird also tatsächlich nie ausgeführt.

Sinnvoller werden `goto`-statements natürlich erst in Verbindung mit anderen Konstrukten, z.B. bei Kombination mit if-Abfragen: `if (`*Bedingung*`) goto` *Zeilennummer*. Das sieht auf den ersten Blick aus wie eine willkommene Flexibilisierung z.B. des `exit`-statements, weil man ein solches goto-Konstrukt auch außerhalb von Schleifen einsetzen kann und gleichzeitig auch den Zielpunkt des Sprungs flexibel wählen und mit angeben kann. Tatsächlich beginnt aber genau damit schon ein Teil der Verwirrung: Die Zeilennummern sind völlig frei wählbar, also muß man entweder akribisch darauf aufpassen, daß sie

z.B. immer in numerisch aufsteigender Reihenfolge im Programmtext erscheinen (was bei Hinzufügen neuer Zeilennummern in echte Arbeit ausarten kann), oder man muß jede Zeilennummer beim Lesen eines Programmtextes langwierig in einer ungeordneten Folge suchen. Nicht zuletzt deshalb sollte man mit goto-Befehlen möglichst sparsam umgehen.

8 Prozeduren

8.1 Sinn von Prozeduren

Prozeduren (Unterprogramme und Funktionen) sind ebenfalls Kontrollstrukturen, aber so wichtige, daß sie ein eigenes Kapitel erhalten. Funktionen sind sozusagen ein Spezialfall von Unterprogrammen; der Unterschied wird weiter unten erklärt. Wenn nicht explizit anders erwähnt, gilt alles für Unterprogramme gesagte auch für Funktionen.

Eine der Grundideen für die Einführung von Prozeduren ist wieder die Vereinfachung von Wiederholungen von Programmteilen – hier jedoch nicht unmittelbar aufeinander folgende Wiederholungen, sondern Wiederholungen desselben Programmabschnitts an mehreren getrennten Stellen im Programmablauf. Aus einem solchen wiederholt gebrauchten Abschnitt kann man eine Prozedur machen, die an allen benötigten Stellen mit einem einfachen, einzelnen Befehl aufgerufen werden kann. Wie schon bei Schleifen erhöht dies die Lesbarkeit des ganzen Programms beträchtlich.

Ein weiterer, wesentlicher Vorteil der Einführung von Prozeduren ist die damit automatisch einhergehende Modularisierung. Prozeduren brauchen eine definierte Schnittstelle mit dem Hauptprogramm (welche Variablen werden übergeben, usw., s.u.) und bearbeiten eine fest umrissene Aufgabe. Das kann offensichtlich Programmänderungen enorm vereinfachen.

Diese beiden Vorteile sind so beträchtlich, daß es sich sogar lohnen kann, Prozeduren einzuführen, die nur ein einziges Mal gebraucht werden, bis schließlich das Hauptprogramm fast nur noch aus einer Folge von Unterprogrammaufrufen besteht. Tatsächlich kann man dies jedoch auch übertreiben, da jeder Unterprogrammaufruf im laufenden Programm einen gewissen internen Verwaltungsaufwand erzeugt, wodurch sich das Programm verlangsamt.

8.2 Unterschied Funktion ↔ Unterprogramm

Der scheinbar einzige relevante Unterschied zwischen Funktionen und Unterprogrammen ist der, daß die Kommunikation zwischen Unterprogramm und rufendem Programm ausschlieslich über die Variablen der Parameterliste erfolgt (s.u.; abgesehen von globalen Variablen, die hier nicht behandelt werden). Vor der Rückkehr zum rufenden Programm muß dagegen dem Funktionsnamen ein Wert zugewiesen worden sein (das Ergebnis der Funktion); daher kann dann der Funktionsname im rufenden Programm in arithmetischen Ausdrücken verwendet werden (wie Variablennamen) und daher muß ihm auch ein Variablentyp zugewiesen werden.

Obwohl dies vom Standard eigentlich nicht gefordert wird, empfiehlt es sich daher, statt einem Unterprogramm genau dann eine Funktion zu verwenden, wenn die Rechnung im Unterprogramm keine Variablen der Parameterliste verändert (wenn diese also nur input sind) und nur einen einzigen (Zahlen)Wert als Ergebnis liefert. Ansonsten sollte man eigentliche Unterprogramme verwenden.

8.3 Aufruf und Rückkehr

Ein Unterprogramm wird folgendermaßen aufgerufen:

```
program haupt
implicit none
real::a,b,c
a=42.0
call blabla(a,b,c)
write(*,*)b,c
end

subroutine blabla(abla,bbla,cbla)
implicit none
real::abla,bbla,cbla
bbla=abla+abla
cbla=sqrt(abla)+bbla
return
end
```

Beim statement `call blabla` springt die Ausführung vom Hauptprogramm ins Unterprogramm `blabla` und wird dort fortgesetzt. Bei einem `return`-statement wird umgekehrt wieder zum rufenden Programm (hier Hauptprogramm) zurückgesprungen; die Ausführung wird mit dem nächsten Befehl (hier write) fortgesetzt. Der eigentliche Unterprogrammteil steht separat vom Hauptprogramm (im gleichen oder aber auch in einem separaten file) und endet wie das Hauptprogramm mit `end`. Damit der compiler versteht, daß es sich um ein Unterprogramm handelt, muß es allerdings mit `subroutine` beginnen.

Der Aufruf einer Funktion sieht in einigen Teilen ähnlich, in einigen Teilen anders aus:

```
program haupt
implicit none
real::a,b,c
real::blabla
a=42.0
b=a+a
c=blabla(a,b)+b
write(*,*)b,c
end

real function blabla(abla,bbla)
implicit none
real::abla,bbla
blabla=sqrt(abla)
return
end
```

Wie oben angedeutet, muß der Funktionsname (hier blabla) im rufenden Programm in einer Typvereinbarung erscheinen (wie ein Variablenname). Zusätzlich muß der Beginn des Funktionstexts, der eigentlich alleine durch `function` signalisiert wird, ebenfalls mit einer Typvereinbarung verbunden werden (hier `real`), die sozusagen auf den dahinterstehenden

Funktionsnamen wirkt. Innerhalb des Funktionstexts muß dem Funktionsnamen ein Wert zugewiesen werden (sinnvollerweise vor dem `return`, und sinnvollerweise das Ergebnis einer Rechnung). Dementsprechend wird die Funktion auch nicht mit einem `call` aufgerufen, sondern direkt durch die Verwendung ihres Namens in einem arithmetischen Ausdruck, wie in diesem Beispiel. (Der output dieses und des vorigen Beispiels sollten identisch sein.)

Natürlich kann ein Unterprogramm (oder eine Funktion) weitere Unterprogramme aufrufen, völlig analog zu den hier gegebenen Beispielen.

8.4 Parameterlisten: einfache Variablen

Abgesehen vom mit dem Funktionsnamen zurückgegebenen Resultat bei Funktionen wird zwischen Hauptprogramm und Unterprogramm über die Variablen in den Parameterlisten kommuniziert (`a,b,c` bzw. `abla,bbla,cbla` in den obigen beiden Beispielen). Wie gezeigt müssen die Variablennamen beim Aufruf (`call`) eigentlich nicht identisch mit denen in der Unterprogrammdefinition sein. Es ist jedoch sehr zu empfehlen, das trotzdem zumindest zu versuchen, soweit es möglich und sinnvoll ist (erhöht die Lesbarkeit).

Auf jeden Fall müssen die *Variablentypen* übereinstimmen: Wenn z.B. die erste Variable in der Parameterliste der Unterprogrammdefinition ein `integer` ist und die zweite ein `character(len=20)`, dann muß in jedem Aufruf die erste Variable der Parameterliste ebenfalls ein `integer` sein und die zweite ein `character(len=20)`. Im Prinzip könnte dies vom compiler fast immer überprüft werden, und er könnte bei Verstössen eine Warnung ausgeben. Da dies in früheren Standards (z.T. mit Absicht) weniger rigoros gehandhabt wurde, kann man sich darauf jedoch nicht verlassen; viele compiler machen eine solche Überprüfung auch nur nach expliziter Aufforderung dazu (mit einer geeigneten compiler-Option). Ein solcher Fehler kann daher ggf. bis ins laufende Programm unentdeckt bleiben, wo er dann entweder zu schwer zu diagnostizierenden Programmabstürzen führt (typischerweise eine sogenannte *segmentation violation*, wenn die Programmkontrolle den ihr eigentlich zugewiesenen Speicherbereich verläßt) oder zu noch schwieriger detektierbaren fehlerhaften Resultaten.

Variablen, die im Unterprogramm vereinbart werden und *nicht* in der Parameterliste erscheinen, sind lediglich lokal innerhalb des Unterprogramms bekannt. Sie verlieren in der Regel ihren Wert beim Verlassen des Unterprogramms; insbesondere kann nicht damit gerechnet werden, daß sie noch ihren „alten“ Wert haben, wenn dasselbe Unterprogramm später erneut aufgerufen wird. Wenn dieses „Erinnerungs“-Verhalten für einige (oder alle) Variablen in einem Unterprogramm erwünscht ist, sind diese Variablen in einem nicht-ausführbaren `save`-statement im Deklarationsteil zu nennen: `save a,b` sorgt dafür, daß die Variablen namens a und b bei einem erneuten Aufruf desselben Unterprogramms denselben Wert haben, den sie beim letzten `return` aus diesem Unterprogramm hatten. Ein `save` ohne Variablennamen ist äquivalent zu einem `save` mit allen im Unterprogramm vorkommenden Variablen.

Man beachte: Die Variablen des rufenden Programms behalten ihren Wert über den Unterprogramm-Aufruf hinweg bei – sofern sie nicht in der Parameterliste des Aufrufs stehen und im Unterprogramm dann tatsächlich verändert werden.

8.5 Parameterlisten: Felder

Natürlich können über Parameterlisten auch ein- und mehrdimensionale Felder übergeben werden:

```
program feld
implicit none
integer,dimension(5,6)::ia
i(1,1)=1
i(2,1)=2
i(3,1)=3
i(1,2)=4
i(2,2)=5
i(3,2)=6
call klein(ia)
end

subroutine klein(ia)
implicit none
integer,dimension(5,6)::ia
write(*,*)ia(3,2)
return
end
```

Hier wird das Feld `ia` übergeben. Es hat im Haupt- und Unterprogramm genau dieselbe, feste Größe; diese Situation ist absolut unproblematisch. Der output dieses Programms ist daher die Zahl 6.

Der Nachteil dieser unproblematischen, festen Größenvereinbarungen ist der, daß das Unterprogramm nur für genau diese Feldgröße eingesetzt werden kann. In der Praxis möchte man aber oft auch allgemeine Unterprogramme schreiben können, die z.B. auch von anderen Leuten in anderen Situationen für ganz andere Feldgrößen verwendbar sind. Dafür kann man in Unterprogrammen variable Feldgrößen vereinbaren:

```
program feld
implicit none
real,dimension(10)::x(10)
real::sum_up
integer::n,i n=5
do i=1,n
  x(i)=real(i)
end do
summe=sum_up(x,n)
end

real function sum_up(x,n)
implicit none
real,dimension(n)::x(n)
real::a
integer::i,n
a=0.0
```

```
do i=1,n
  a=a+x(i)
end do
sum_up=a
return
end
```

Die Funktion in diesem Beispiel summiert die ersten `n` Elemente des Vektors `x` auf, ist jedoch so allgemein geschrieben, daß sie nicht nur in diesem Beispiel, also für $n = 5$, verwendet werden kann, sondern auch in anderen Programmen für $n = 1000$ oder ganz andere n-Werte einsetzbar ist. Die tatsächliche variable Feldgröße muß in der Parameterliste mit an das Unterprogramm übergeben werden, wie hier im Beispiel; auch dann, wenn sie nirgendwo anders auftaucht als eben in der Deklaration des variablen Feldes im Unterprogramm.

Im alten Fortran77-Standard ist es nicht möglich, im Hauptprogramm variable Felder zu vereinbaren, da dort zu Beginn des Programms der maximal verfügbare Speicherplatz festliegen muß. Daher müssen innerhalb dieses Standards die Feldvereinbarungen im Hauptprogramm fest sein und mindestens so groß, wie die größte Feldgröße tatsächlich sein wird, wenn das Programm läuft. (Im obigen Beispiel dürfte man z.B. im Hauptprogramm den Wert von `n` nicht größer als 10 wählen, oder man müßte in der Deklaration des Feldes `x` seine feste Länge entsprechend größer wählen.)

In Fortran90/95 entfallen diese Beschränkungen, und es gibt eine Vielzahl weiterer Möglichkeiten, Felder variabler Größe in verschiedenen Programmteilen zu vereinbaren. Für die Zwecke dieses Kurses ist die hier vorgestellte Konstruktion jedoch ausreichend.

Mehrdimensionale variable Felder bringen ein weiteres Problem mit sich: Die folgende **falsche(!)** Variante des weiter oben schon aufgeführten Beispiels liefert als output nicht den Wert 6, sondern ein undefiniertes (compilerabhängiges) Resultat (in der Regel den Wert 4):

```
program feld
implicit none
integer,dimension(5,6)::ia
integer::m,n
i(1,1)=1
i(2,1)=2
i(3,1)=3
i(1,2)=4
i(2,2)=5
i(3,2)=6
m=3
n=2
call klein(ia,m,n)
end

subroutine klein(ia,m,n)
implicit none
integer::m,n
intege,dimension(m,n)::ia
write(*,*)ia(3,2)
```

```
return
end
```

Der Grund dafür ist die Art und Weise, in der Fortran Felder abspeichert und Parameterlisten behandelt: Der Speicher des Computers ist aus der Sicht des Programms nur eindimensional adressierbar. Also muß auch ein mehrdimensionales Feld in einer eindimensionalen Reihenfolge abgespeichert werden. In Fortran werden dazu die Feldelemente so abgespeichert, daß der erste Feldindex seinen Wert am schnellsten erhöht und die nachfolgenden Indices jeweils langsamer. In diesem Beispiel sorgt die Feldvereinbarung im Hauptprogramm für die Bereitstellung einer Speicherregion, in der die einzelnen Elemente des Felds `ia` in folgender Reihenfolge abgelegt werden: `ia(1,1)`, `ia(2,1)`, `ia(3,1)`, `ia(4,1)`, `ia(5,1)`, `ia(1,2)`, `ia(2,2)`, ..., da die (feste) Obergrenze des ersten Index den Wert 5 hat. Das Feldelement `ia(3,2)` ist also „eigentlich“ das achte(!) Element einer eindimensionalen Reihenfolge.

Bei der scheinbaren Übergabe des Feldes `ia` vom Hauptprogramm ins Unterprogramm wird nicht etwa die gesamte Struktur des Feldes mit übergeben (so wie sie im Hauptprogramm vereinbart wurde). Es werden auch nicht die aktuellen Werte auf den Elementen des Feldes übergeben oder alle ihre Speicheradressen. Tatsächlich übergeben wird formal nur die *Startadresse* des Speicherbereichs, der im Hauptprogramm (implizit, in der Deklaration) dem Feld `ia` zugeordnet wurde. An derselben Stelle fängt dann im Unterprogramm ein Feld `ia` an, das hier jedoch nicht notwendigerweise in derselben Weise vereinbart sein muß. (Wie oben gesagt, müßte eigentlich der Feldname im Unterprogramm nicht derselbe sein wie im Hauptprogramm. Da auch bei einfachen Variablen eine Speicheradresse übergeben wird, müßte an dieser Stelle der Parameterliste noch nicht einmal zwangsläufig ein Feld vereinbart werden. Aber diese möglichen Abweichungen von der eigentlichen Logik sind praktisch nie wirklich nötig und noch weniger nützlich oder übersichtlich.)

Im Unterprogramm ist das Feld `ia` beim tatsächlichen Programmlauf so vereinbart, also ob dort eine Deklaration `integer ia(3,2)` stünde (aufgrund der Festlegungen `m=3` und `n=2` im Hauptprogramm, die dann im Unterprogramm als Feldgrenzen in der Deklaration verwendet werden). Wieder ist die Speicherreihenfolge jedoch die formal gleiche, eindimensionale – in diesem Fall aber aufgrund der anderen Index-Obergrenzen aktuelle eine etwas andere: `ia(1,1)`, `ia(2,1)`, `ia(3,1)`, `ia(1,2)`, `ia(2,2)`, ...; das Feldelement `ia(3,2)` ist in dieser Reihenfolge also das sechste(!). Da aber nur die Speicher-Anfangsadresse übergeben wurde, hat sich dadurch an der Anordnung der eigentlichen Zahlen im Speicher (und an der Zugriffsreihenfolge auf sie) nichts geändert – also wird de facto im Unterprogramm mit `ia(3,2)` auf das sechste Element der eindimensionalen Speicherreihenfolge des *Hauptprogramms* zugegriffen, und das ist im Hauptprogramm das Element `ia(1,2)=4` (wenn automatische Programm-Optimierungen durch den compiler das nicht noch weiter „durcheinandergebracht“ haben).

Trotzdem muß man nicht auf variable, mehrdimensionale Felder verzichten, wenn man sich nur dieser Problematik bewußt ist. Der saubere Ausweg aus dieser Situation ist folgender:

```
program feld
implicit none
integer,dimension(5,6)::ia
integer::m,n,k
i(1,1)=1
i(2,1)=2
```

```
i(3,1)=3
i(1,2)=4
i(2,2)=5
i(3,2)=6
m=3
n=2
k=5
call klein(ia,m,n,k)
end

subroutine klein(ia,m,n,k)
implicit none
intege::m,n
integer,dimension(k,n)::ia
write(*,*)ia(3,2)
return
end
```

Dieses Programm liefert wieder den korrekten Wert 6 als output. Der Trick besteht darin, die tatsächliche Obergrenze des schneller laufenden (ersten) Index als zusätzliche Variable mit an das Unterprogramm zu übergeben und dort in der Deklaration des variablen Feldes zu verwenden (bei Feldern mit mehr als zwei Dimensionen natürlich entsprechend mehrere Index-Grenzen). Dadurch wird erzwungen, daß die formale Abspeicherungsreihenfolge im Unterprogramm dieselbe ist wie im Hauptprogramm. Trotzdem bleibt das Unterprogramm so allgemein wie vorher auch.

Dieser Trick wird in professionellen Unterprogramm-Bibliotheken auch häufig verwendet. Aus der hier gegebenen Erklärung sollte verständlich werden, was mit dem dabei scheinbar überflüssigen, zusätzlichen Parameter (hier `k`) eigentlich gemeint ist. In Beschreibungen von subroutinen-Bibliotheken heißt dieser Parameter üblicherweise „leading dimension“, weil er angibt, welchen Maximalwert der erste Index (die „führende Dimension“) in der Vereinbarung eines Feldes (hier `ia`) im rufenden Programm hat; das ist die wesentliche Information, die das gerufene Unterprogramm braucht, um das variable Feld korrekt zu vereinbaren. Der Maximalwert des zweiten Index eines zweidimensionalen Feldes ergibt sich indirekt aus dem bekannten Maximalwert des ersten Index und der in normalen Anwendungsfällen ohnehin immer mit übergebenen Gesamtgröße des Feldes und muß daher nicht auch noch übergeben werden. (Drei- oder mehrdimensionale Felder kommen in Bibliotheksroutinen nur selten vor.)

8.6 Parameterlisten: optionale Argumente

Nach den bisherigen Erläuterungen müssen die Parameterlisten des Prozeduraufrufs und der Prozedur selber genau übereinstimmen, nach Anzahl, Reihenfolge und Art der Parameter. Dies entspricht dem Fortran77-Standard. Im Fortran90/95-Standard kommt die Möglichkeit von sogenannten *optionalen Argumenten* hinzu. Diese werden in einigen eingebauten Fortran90/95-Prozeduren verwendet und deshalb hier kurz erläutert.

Optionale Argumente können beim Aufruf in der Parameterliste erscheinen, müssen aber nicht vorhanden sein. Damit der compiler trotzdem korrekt zuordnen kann, welcher der möglichen Parameter bei einem Aufruf gemeint ist, müssen solche optionalen Parameter

mit ihrem in der Prozedurdefinition gegebenen, eindeutigen Namen bezeichnet werden, in der folgenden Art:

```
call routine(name1=x,name3=y)
```

Hier soll also die Variable `x` als optionales Argument mit dem Namen `name1` und die Variable `y` als optionales Argument mit dem Namen `name3` übergeben werden. Aus der Prozedurdefinition oder -beschreibung muß hervorgehen, wie diese Argument-Namen lauten müssen und welche der optionalen Argumente vorhanden sein müssen oder fehlen dürfen.

8.7 Felder variabler Größe im Hauptprogramm

Im alten Fortran77-Standard konnten, wie oben gezeigt, in Unterprogrammen Felder variabler Größe verwendet werden (Voraussetzung: das Feld und seine aktuellen Indexgrenzen (und ggf. die „leading dimension“) werden per Parameterliste vom Hauptprogramm an das Unterprogramm übergeben); im Hauptprogramm mußten jedoch alle Felder feste Index-Obergrenzen haben, die zum Zeitpunkt der Programm-compilierung bekannt sind und sich dann von Programmlauf zu Programmlauf nicht mehr ändern können.

In Fortran90/95 gibt es zahlreiche Möglichkeiten, variable Felder zu definieren und zu verwenden, insbesondere auch in Unterprogrammen. Hier soll nur eine Möglichkeit erwähnt werden, die die Verwendung variabler Felder auch im Hauptprogramm erlaubt:

- Deklaration des Feldes mit dem Zusatzattribut `allocatable`;
  gleichzeitig Ersatz der festen Index-Obergrenzen durch je einen Doppelpunkt;
- sobald die Index-Obergrenzen im Programmablauf bekannt sind (z.B. durch Eingabe ihrer Werte) und vor der ersten Verwendung des Feldes, wird der jeweils tatsächlich nötige Speicherbereich durch eine `allocate`-Anweisung reserviert;
- nach Verwendung des Feldes muß der Speicherbereich durch ein `deallocate` wieder freigegeben werden.

Sinnloses Beispiel mit einem zweidimensionalen Feld:

```
program variables_feld
implicit none
integer m,n
real,allocatable,dimension(:,:)::b
...
read(1,*)m,n
allocate(b(m,n))
do i=1,m
  do j=1,n
    b(i,j)=sin(sqrt(real(i+j)))
  end do
end do
...
deallocate(b)
...
end
```

Der saubere Gebrauch von `allocate` und `deallocate` sollte eigentlich auch eine (eigentlich optionale) Statusabfrage umfassen, mit der festgestellt werden kann, ob die Speicher(de)allokation erfolgreich war oder nicht. Damit läßt sich z.B. der Fall sauber abfangen, daß ein `allocate` mehr Speicher fordert als noch zur Verfügung steht. Dies wird über das optionale `integer`-Argument `stat` geregelt, das bei erfolgreicher (De)Allokation den Wert Null erhält:

```
program variables_feld2
implicit none
integer k,status
real,allocatable,dimension(:)::c
...
read(1,*)k
allocate(c(k),stat=status)
if (status /= 0) then
  write(*,*)'Fehler bei Speicherzuweisung für Feld c!'
  stop
end if
do i=1,k
  c(k)=real(k)
end do
...
deallocate(c)
...
end
```

8.8 Verwendung von Bibliotheksroutinen

Außer selbstgeschriebenen Prozeduren und mit dem compiler mitgelieferten Prozeduren kann man auch weitere Prozeduren verwenden, die in sogenannten (subroutinen-)Bibliotheken bereitgestellt werden. Im Kurs stehen dafür die frei verfügbaren Standardbibliotheken BLAS, LAPACK und SLATEC zur Verfügung. BLAS (= basic linear algebra subroutines) enthält grundlegende Operationen der linearen Algebra (z.B. Produkte zwischen Vektoren und Matrizen). Viele wissenschaftliche Programme verwenden solche Operationen sehr häufig; daher gibt es maschinenangepaßte Varianten dieser BLAS-Bibliothek, die die Ausführung solcher Programme erheblich beschleunigen können. LAPACK (= linear algebra package) enthält u.a. Unterprogramme zur Lösung von linearen Gleichungssystemen und Eigenwertproblemen und greift intern extensiv auf BLAS zurück. SLATEC ist eine Art Universalbibliothek für fast alle Zwecke; hier im Kurs wird sie für die Fälle verwendet, die von BLAS/LAPACK nicht abgedeckt werden.

Alle drei Programmpakete sind „beheimatet“ bei `http://www.netlib.org/`. Sie können von dort heruntergeladen werden (einzeln oder im Gesamtpaket), und dort findet man (im Prinzip) auch eine komplette Dokumentation. Zur Benutzung von LAPACK ist es empfehlenswert, zunächst im Lapack-Users'-Guide `http://www.netlib.org/lapack/lug/` die benötigte Routine zu suchen; dort ist aufgeführt,

- welche Prozeduren zur Verfügung stehen, geordnet nach Anwendungsbereichen (lineare Gleichungssysteme, Eigenwertprobleme, usw.),
- die grundlegenden Prinzipien jedes Bereichs,

- die (stringente) Nomenklatur zur Benennung der Unterprogramme,
- und Kurzerläuterungen zu den einzelnen Unterprogrammen.

Anfängern sei dringend empfohlen, sich zunächst an die sogenannten „driver routines“ zu halten; sie lösen in aller Regel das vorliegende Problem komplett, indem sie intern eine geeignete Abfolge sogenannter „computational routines“ aufrufen und so den Benutzer nicht mit einer Überfülle von Details belästigen. Die direkte Benutzung der „computational routines“ erlaubt eine genauere Steuerung und Eingriffe in einzelne Zwischenschritte, erfordert aber größere Detailkenntnisse. Hat man eine geeignete „driver routine“ gefunden, sollte man deren genaue Beschreibung im Abschnitt „individual routines“ von `http://www.netlib.org/lapack/` aufsuchen. Dort findet man

- wie man die subroutine aufrufen muß,
- welche Parameter sie braucht und welche sie ausgibt,
- und wie diese Parameter im rufenden Programm vereinbart werden müssen.

All dies ist jedoch extrem kurz gehalten, sodaß man sich ohne gründliches Vorwissen kaum zurechtfinden wird.

Alle diese Routinen können in eigenen Programmen wie normale intrinsische Routinen aufgerufen werden, wenn man den Compiler-Befehl etwas erweitert zu:

```
gfortran -o MeinProgramm MeinProgramm.f90 -llapack -lblas
```

(für den Fall, daß man LAPACK verwenden möchte; wenn man nur BLAS alleine braucht, kann man natürlich auf die Angabe von `-llapack` verzichten.)

Die Dokumentation für SLATEC ist irgendwo zwischen kryptisch und nicht vorhanden anzusiedeln. Daher ist zu empfehlen, für eine erste Navigation auf GAMS zurückzugreifen (guide to available mathematical software, `http://gams.nist.gov/`). Dort findet sich neben einer Suchfunktion auch eine einigermaßen übersichtliche, baumartige Aufteilung nach Problemklassen. Als Nebeneffekt erhält man dabei nicht nur eine Übersicht über den Inhalt von SLATEC, sondern gleichzeitig auch über viele andere Bibliotheken.

9 Programmierdisziplin!

- Warum kein „Freistil-Programmieren“?
  - Programme anderen verständlich machen
  - eigene Programme später wieder verstehen
  - Schreiben längerer Programme erleichtern
- „*Strukturiertes Programmieren*“ = Programme übersichtlich und verständlich gestalten; insbesondere:
  - sinnvolle Variablennamen (Längenbeschränkung ist antik)
  - gleiche Variablen in verschiedenen Programmblöcken gleich nennen
  - längere, abgeschlossene Unteraufgaben zu subroutines machen, auch wenn nur einmal verwendet
  - do- und if-Blöcke usw. systematisch und immer gleich einrücken
  - veraltete, unübersichtliche Programmstrukturen (z.B. `go to`) vermeiden
- „*Portabilität*“: keine Abweichungen vom Standard, insbesondere:
  - alle Variablen initialisieren;
  - für lokale Variablen, deren Werte über ein Verlassen und Wiedereintritt in eine subroutine erhalten bleiben sollen, `save`-Anweisung verwenden.
- *Kommentare!* Natürlich keine trivialen, aber im Zweifelsfall immer viel mehr als man zunächst denkt.
- Tips zur *Programmentwicklung:*
  - modular! = Niemals ein Programm einfach von Anfang bis Ende herunterschreiben, sondern jeden sinnvollen Abschnitt sofort testen. Nachträgliche Fehlersuche immer viel schwerer.
  - erleichternde Hilfsmittel ausnutzen, z.B.: compiler-Optionen zur Feldgrenzenüberwachung, Variablentyp-Kollisionen, Beeinflussung des Verhaltens bei „floating point exceptions“, usw.
- Verzeichnisse übersichtlich halten:
  - Extra-file mit Verzeichnis-Inhaltsangabe
  - nicht mehr benötigte Programmvarianten sofort löschen
  - sinnvolle file-Namen: je länger, desto besser; Varianten eines Programms und Zusammengehörigkeit Programm/Input/Output sollte erkennbar sein.

10 Wichtige, aber hier nicht behandelte Dinge

Die vorliegende Kurzbeschreibung läßt zahlreiche, wichtige Sprachelemente unbehandelt. Dies sind einige, bereits in Fortran77 realisierte Dinge, wie u.a.:

- globale Variable über common-Blöcke
- Parameter
- include-statement
- viele weitere Feinheiten von I/O:
  - Formatierung
  - implizite Schleifen
  - file-Status-Abfrage
  - end/err-Behandlung
  - Positionierung mit rewind und backspace
  - random access (statt sequential)

Im neueren Fortran90/95-Standard kommt eine Vielzahl weiterer Dinge dazu, u.a.:

- weitere Kontrollstrukturen (z.B. case-statement)
- Module (für globale Variable, explizite Prozedur-Interfaces)
- viele Varianten variabler Felder
- array-Operationen
- keyword-Argumente und optionale Argumente
- pointer
- ...

Dafür wird auf einschlägige Lehrbücher verwiesen.

Index

Anweisung, 5
Argumente
    optionale, 26
arrays, siehe Felder, 11

Befehl, 5
Befehle
    ausführbare, 5
Bibliotheken, 28

`call`, 21
compiler, 4
    Optionen, 4

Datei, 13
    ausführbare, 4
    Programm, 4
dimension
    leading, 26
`do`, 17

`exit`, 18

Felder, 11
    Index, 11
    variable, 23, 27
file, 13
    Name, 14
Format, 15
Funktion, 20, 21
Funktionen
    eingebaute, 12
    generische, 12

`goto`, 19

i/o-unit, 13
`if`, 16
`implicit none`, 10

Kommentar, 6
Kommentarzeile, 6
Konstante, 7

leading dimension, 26

Matrizen, siehe Felder, 11

`open`, 13

Parameterliste, 22
`program`, 6
Prozedur, 20

`return`, 21

Schleife, 17
    while, 18
    while/until, 18
    Zähl-, 18
Sprungbefehl, 19
statement, 5

Typumwandlung, 10

unit, 13
    number, 13
Unterprogramm, 20

Variable, 7
    Deklaration, 8
    global, 20
    Initialisierung, 7
    integer, 8
    lokal, 22
    Name, 7
    real, 8
    Text, 9
    Typen, 8
      implizit, 9
      Umwandlung, 10
Vektoren, siehe Felder, 11

Zeilenformat
    festes, 6
    freies, 6