

- Rechenzentrum -

Einführung in die FORTRAN Programmierung

Universität Osnabrück
- Rechenzentrum -
Albrechtstraße 28 (AVZ)
D-4500 Osnabrück

Datum: 97/01/28
Version: 1

Inhaltsverzeichnis:

1.	Einleitung	4
	1.1. Literaturhinweise	
	
	4	
	1.2. Typografische Konventionen	
	
	4	
2.	Der Entwicklungsprozeß im Überblick	5
	2.1. Die einzelnen Schritte	
	
	5	
	2.2. Klären der Problemstellung	
	
	5	
	2.3. Entwickeln einer Lösung	
	
	5	
	2.4. Entwerfen eines Algorithmus	
	
	5	
	2.5. Schreiben eines Programms	
	
	6	
	2.6. Implementieren eines Programms	
	
	6	
3.	Was ist ein Algorithmus?	7
	3.1. Algorithmen im täglichen Leben	
	
	7	
	3.2. Beispiel 1 - Münzfernsprecher	
	
	7	
	3.3. Eigenschaften	
	
	7	
	3.4. Beispiel 2 - Summe der Zahlen von 1 bis N	
	
	8	
	3.5. Bausteine eines Algorithmus	
	
	8	
	3.6. Darstellungsformen	
	
	9	
	3.7. Beispiel 3 - Flußdiagramm	
	
	10	
	3.8. Beispiel 4 - Pseudo-Code	
	
	10	
4.	Wie führt ein Computersystem ein Programm aus?	12
	4.1. Arbeitsweise eines Computersystems	
	
	12	

4.2. Quell- und Zielprogramm
12	
4.3. Höhere Programmiersprachen
13	
4.4. Geschichte von FORTRAN
14	
5. Überblick über die FORTRAN Sprachelemente.....	15
5.1. Vergleich mit einer natürlichen Sprache
15	
5.2. Schreibweise, Syntax und Semantik
15	
5.3. Zeichen
15	
5.4. Wörter
16	
5.5. Ausdrücke
16	
5.6. Anweisungen
16	
5.7. Programm-Einheit
17	
5.8. Programm
17	
6. Elementare Sprachelemente.....	18
6.1. Literale Konstanten
18	
6.2. Symbolische Konstanten
22	
6.3. Variablen
23	
6.4. Arithmetische Ausdrücke
24	
6.5. Beispiel 5 - Auswertungsreihenfolge für einen Ausdruck
24	
6.6. Beispiel 6 - Automatische Umwandlung
25	
6.7. Arithmetische Zuweisungen
25	
6.8. Beispiel 7 - Kreisberechnung
25	

6.9. Beispiel 8 - Variablen in Zuweisungen
25	
6.10. Beispiel 9 - Genauigkeitsverlust
26	
6.11. Listengesteuerte Ein- und Ausgabe
26	
6.12. Beginn und Ende eines Programmes
27	
6.13. Beispiel 10 - Kreisberechnung
27	
6.14. Zusammenfassung
28	
7. Festes Eingabeformat und Kommentare	29
7.1. Aufteilung der verfügbaren Spalten
29	
7.2. Kommentare
30	
7.3. Beispiel 12 - Kommentare
30	
7.4. Zusammenfassung
30	
8. Wiederholungen, Verzweigungen und Sprünge	31
8.1. DO Anweisungen (Wiederholungen)
31	
8.2. Felder
32	
8.3. Beispiel 15 - Felder in verschachtelten DO Anweisungen
33	
8.4. Logische Ausdrücke
34	
8.5. Vergleiche
34	
8.6. IF Anweisungen (Verzweigungen)
35	
8.7. Sprungmarken und GOTO Anweisungen
36	
8.8. PAUSE Anweisungen und STOP Anweisungen
37	
8.9. Zusammenfassung
37	

9. Formelfunktionen, Prozeduren und COMMON Blöcke.....	38
9.1. Formelfunktionen	
.....	
38	
9.2. Beispiel 17 - Berechnung des Kugelvolumens	
.....	
38	
9.3. Funktionen	
.....	
39	
9.4. Vordefinierte Funktionen	
.....	
40	
9.5. Beispiel 18 - Satz des Pythagoras	
.....	
41	
9.6. Beispiel 19 - Schaltjahr	
.....	
41	
9.7. Unterprogramme	
.....	
41	
9.8. Beispiel 20 - Kreisberechnung	
.....	
42	
9.9. Prozeduren mit variablen Feldern und Zeichenketten	
.....	
43	
9.10. Beispiel - Vektormultiplikation	
.....	
44	
9.11. Prozeduren mit statischen Variablen	
.....	
44	
9.12. COMMON Blöcke	
.....	
45	
9.13. Beispiel 21 - Globale Variablen in einem COMMON Block	
.....	
45	
9.14. Zusammenfassung	
.....	
47	
10. Formatierte Ein- und Ausgabe und Dateiverarbeitung.....	50
10.1. Formatierte Ein- und Ausgabe	
.....	
50	
10.2. Beispiel 22 - Ausgeben einer Tabelle	
.....	
51	
10.3. Dateiverarbeitung	
.....	
52	
10.4. Beispiel 23 - Einlesen aus einer Datei	
.....	
53	
10.5. Zusammenfassung	
.....	
54	

11.	Bemerkungen.....	55
	11.1. Programmiermethodik	
	
	55	
	11.2. Weitere Sprachelemente	
	
	55	
	11.3. Wünschenswerte Erweiterungen	
	
	56	
	11.4. Weitere Werkzeuge	
	
	56	
	11.5. Weitere Bibliotheken	
	
	56	
12.	Übungen.....	57
	12.1. Übung 1: Algorithmen, Flußdiagramme und Pseudo-Code	
	
	57	
	12.2. Übung 2: Entwicklungsumgebung	
	
	58	
	12.3. Übung 3: Konstanten, Variablen, Ausdrücke und Zuweisungen	
	
	59	
	12.4. Übung 4: DO Anweisungen und Felder	
	
	62	
	12.5. Übung 5: Logische Ausdrücke und Zuweisungen, IF Anweisungen	
	
	63	
	12.6. Übung 6: Formelfunktionen und Funktionen	
	
	64	
	12.7. Übung 7: Unterprogramme und COMMON Blöcke	
	
	65	
	12.8. Übung 8: Ein- und Ausgabe und Dateiverarbeitung	
	
	66	
13.	Anhang A: Wichtige CMS Kommandos	67

1. Einleitung

Dieses Skript dient Ihnen als ausformulierter Begleittext zum Kurs "Einführung in die FORTRAN Programmierung" des Rechenzentrums und kann ggfs. auch als Tutorial zum Selbststudium genutzt werden.

Im Skript werden die **Grundlagen der Programmiersprache FORTRAN 77** behandelt, die schwerpunktmäßig zur Entwicklung naturwissenschaftlich-technischer Programme verwendet wird. Die Sprache FORTRAN wird nicht mit letzter formaler Präzision eingeführt, da der Verfasser vollständige Syntaxdiagramme eher für abschreckend denn für hilfreich hält. In Zweifelsfällen müssen Sie allerdings auf die offizielle Beschreibung des Sprachumfangs [1] zurückgreifen.

Im Skript wird die Entwicklungs- und Systemumgebung nur am Rande behandelt, da der Schwerpunkt auf den portablen Sprachelementen von FORTRAN liegen soll. Im Rahmen des Skriptes dient die Entwicklungsumgebung auf dem System IBM 3090 unter VM/CMS als Grundlage. Sie besteht aus dem Editor XEDIT, dem FORTRAN Compiler FORTVS2, dem Programm-Lader LOAD und dem Programm-Starter START.

1.1. Literaturhinweise

Eine vollständige Beschreibung des FORTRAN Sprachumfangs finden Sie in:

- [1] FORTRAN77 Sprachumfang für DEC/IBM/Siemens (RRZN)

Eine einführende Beschreibung der Entwicklungsumgebung für das System IBM 3090 unter VM/CMS finden Sie in:

- [2] FORTRAN Programmierung unter VM/CMS (RZUNIOS)

Eine vertiefende Darstellung der Sprache FORTRAN 77 finden Sie z.B. in folgenden Büchern:

- [3] Effective FORTRAN 77,
Michael Medcalf,
Clarendon Press, Oxford, 1985
- [4] Programming in Standard FORTRAN 77,
A. Balfour, D.H. Marwick,
Heinemann Educational Books, London, 1979

1.2. Typografische Konventionen

Im Fließtext werden Textauszeichnungen mit **Fettschrift** verwendet, um Definitionen oder wichtige Textpassagen zu kennzeichnen.

Für Programm-Beispiele wird Schreibmaschinenschrift verwendet. Darüberhinaus sind vollständige Beispiele durch einen Kasten hinterlegt.

Für Syntaxdiagramme wird ebenfalls **Fettschrift** verwendet. Syntaxdiagramme sind durch einen grau hinterlegten Kasten besonders gekennzeichnet. In Syntaxdiagrammen bezeichnen eckige Klammern [...] optionale Syntaxelemente und drei Pünktchen ... Wiederholungen.

2. Der Entwicklungsprozeß im Überblick

In diesem Kapitel lernen Sie den prinzipiellen Ablauf der Programmentwicklung kennen. Die einzelnen Schritte werden in den folgenden Kapiteln detailliert behandelt.

2.1. Die einzelnen Schritte

Bei der Entwicklung eines Programms handelt es sich um einen Prozeß, den Sie in folgenden Schritten durchführen:

1. Klären der Problemstellung
2. Entwickeln einer Lösung
3. Entwerfen eines Algorithmus
4. Schreiben eines Programms
5. Implementieren des Programms

In der Praxis können und werden Sie, abhängig von der Komplexität des Problems und Ihrer Erfahrung im Programmieren, einzelne Prozeßschritte wie z.B. Entwickeln einer Lösung und Entwerfen eines Algorithmus zusammenfassen.

2.2. Klären der Problemstellung

Sie klären zunächst als Ausgangspunkt des Entwicklungsprozesses die genaue **Problemstellung**. Sie stammt in der Regel aus einem Anwendungsgebiet (z.B. Numerik, Statistik, Buchhaltung, Grafik) und soll mit Hilfe eines Computersystems gelöst werden.

In der Physik könnte Ihre Problemstellung z.B. darin bestehen, den Wert des Integrals für die Funktion $\exp(-x^2)$ über ein Intervall $[a,b]$ zu berechnen, wobei die Stammfunktion nicht bekannt ist.

2.3. Entwickeln einer Lösung

Sie entwickeln für die Problemstellung eine modellhafte **Lösung**, in der Sie Voraussetzungen, Eingangs- und Ausgangsparameter und den prinzipiellen Lösungsweg beschreiben.

Für das oben genannte Problem könnte eine einfache Lösung z.B. darin bestehen, das Integral numerisch durch Treppensummen zu approximieren.

2.4. Entwerfen eines Algorithmus

Sie formulieren die Lösung in einem formalen **Algorithmus**. In ihm wird der Lösungsweg in überschaubaren Einheiten (Teilschritten) dargestellt.

Für das oben genannte Problem könnte ein Algorithmus z.B. aus den Teilschritten "Einlesen der Intervallgrenzen", "Definieren einer Folge von Schrittweiten h ", "Berechnen der Treppensummen in Abhängigkeit von h " bestehen:

1. Lies die Intervallgrenzen a und b ein.
2. Lies die anfängliche Schrittweite h ein.
3. Berechne die Treppensumme.
4. Verkleinere die Schrittweite und berechne die Treppensumme erneut.
5. Brich die Berechnung ab, wenn die gewünschte Genauigkeit erreicht ist.

2.5. Schreiben eines Programms

Sie schreiben den Algorithmus in ein **Quellprogramm** einer höheren (problemorientierten) Programmiersprache um, indem Sie die Anweisungen des Algorithmus durch entsprechende Anweisungen der Programmiersprache ausdrücken.

Für das oben genannte Problem könnte es sich z.B. um ein FORTRAN Programm NUMINT handeln, das im Dialog mit dem Benutzer die Daten für die Intervallgrenzen a und b einliest, das Integral approximativ berechnet und als Ergebnis den berechneten Wert liefert:

```
PROGRAM NUMINT
REAL rA, rB, rC
PRINT *, "Geben Sie die Intervallgrenzen a und b ein:"
READ *, rA, rB
...
PRINT *, "Die appr. Auswertung des Integrals ergibt:: ", rC
END
```

2.6. Implementieren eines Programms

Sie übersetzen das Quellprogramm auf einem geeigneten Computersystem mit Hilfe eines Sprachübersetzers (Compiler) in ein **Zielprogramm (ausführbares Programm)**, das anstelle der Anweisungen der Programmiersprache die korrespondierenden Maschinenbefehle enthält und vom Computersystem ausgeführt werden kann.

Für das oben genannte Problem könnte es sich z.B. um ein System IBM 3090/VF mit dem Betriebssystem VM/CMS handeln, das besonders für rechenintensive Programme ausgelegt ist und mit dem deshalb das Integral schnell und genau berechnet werden kann:

Die Transformation des Quellprogramms in ein ausführbares Programm führen Sie mit folgenden Kommandos durch:

```
XEDIT NUMINT FORTRAN      /* startet den Editor XEDIT */
FORTVS2 NUMINT            /* startet den Compiler FORTVS2 */
```

Die interaktive Ausführung des Programms starten Sie anschließend mit folgenden Kommandos:

```
LOAD NUMINT                /* lädt das Programm NUMINT */
START *                    /* startet das geladene Programm */
```

Sie werden vom Programm durch eine Bildschirmmeldung aufgefordert, Werte für die Intervallgrenzen a und b einzugeben. Geben Sie nun die Werte 0 und 1 über die Tastatur ein:

```
Geben Sie die Intervallgrenzen a und b ein:"
0
1
```

Das Programm berechnet auf Grundlage der eingegebenen Werte das Integral approximativ und zeigt das Ergebnis der Berechnung am Bildschirm an:

```
Die appr. Auswertung des Integrals ergibt: 1.0
```

Sie werden festgestellt haben, daß Sie während des Entwicklungsprozesses mehrere Entscheidungen treffen müssen (Lösungsweg, Programmiersprache, Computersystem, ...) und ggfs. auch Prozeßschritte wiederholen müssen, wenn sich eine Entscheidung als falsch herausgestellt hat (Lösung zu ungenau, Programmiersprache ungeeignet, Computersystem zu langsam, ...).

3. Was ist ein Algorithmus?

In diesem Kapitel lernen Sie den Begriff Algorithmus kennen, der in der EDV eine zentrale Bedeutung besitzt.

3.1. Algorithmen im täglichen Leben

Ein **Algorithmus** ist eine formalisierte Beschreibung einer Problemlösung. Eine intuitive Vorstellung von einem Algorithmus geben Ihnen verwandte Begriffe aus dem täglichen Leben:

- Anleitung
- Rezept
- Verfahren
- Vorschrift
- Handlungsaufforderung
- Beschreibung

3.2. Beispiel 1 - Münzfernsprecher

Wir wollen zunächst einen einfachen Algorithmus betrachten, der die Bedienung eines Münzfernsprechers beschreibt:

1. Handapparat abnehmen.
2. Wählton und Speicherbeleuchtung abwarten.
3. Mindestens 3 * 10 Pfennig einwerfen.
4. Wählen.

Dieser Algorithmus (Bedienungsanleitung) zeichnet sich durch folgende Punkte aus:

- Er ist anweisungsorientiert, und die beschriebenen Anweisungen sind ausführbar.
- Er besitzt einen eindeutigen Start (1.) und ein eindeutiges Ende (4.). Die Anweisungen zwischen Start und Ende werden schrittweise ausgeführt.
- Er benutzt ein einheitliches Abstraktionsniveau, das der Komplexität der Anweisungen und dem Erfahrungshorizont der potentiellen Benutzer angemessen ist.

3.3. Eigenschaften

Zusätzlich zu den genannten Eigenschaften müssen Sie in einem Algorithmus berücksichtigen, daß ein Problem in endlicher Zeit gelöst werden muß. Deshalb muß ein Algorithmus nach endlich vielen Schritten beendet werden. Insgesamt ergeben sich somit folgende Anforderungen an einen Algorithmus:

1. **Ausführbarkeit**
2. **Allgemeinheit**
3. **Korrektheit**
4. **Endlichkeit**

Ausführbarkeit:

Ein Algorithmus enthält ausführbare Anweisungen, die die schrittweise Lösung eines Problems beschreiben. Die Anweisungen sind unmißverständlich und handlungsorientiert. Ein Algorithmus verwendet eine einheitliche Abstraktionsebene und verweist ggfs. auf elementarere Algorithmen.

Allgemeinheit:

Ein Algorithmus benennt die Voraussetzungen, unter denen er durchführbar ist. Ein Algorithmus löst eine Klasse von Problemen, nicht ein spezielles Problem. Die Auswahl eines bestimmten Problems erfolgt durch Eingabeparameter, die den weiteren Ablauf des Algorithmus steuern.

Korrektheit:

Ein Algorithmus prüft die Gültigkeit der Eingabeparameter und (Zwischen-) Ergebnisse. Er führt für alle möglichen (also auch unzulässige) Eingabeparameter zu einem sinnvollen Ergebnis, z.B. auch zu einer Fehlermeldung.

Endlichkeit:

Ein Algorithmus besitzt einen definierten Startpunkt, einen definierten Endpunkt und dazwischen endlich viele Anweisungen. Anweisungen innerhalb des Algorithmus werden nur endlich oft durchlaufen. Aus diesem Grund wird der Endpunkt nach endlich vielen Schritten erreicht.

3.4. Beispiel 2 - Summe der Zahlen von 1 bis N

Wir betrachten folgenden Algorithmus zur Summation der Zahlen von 1 bis N:

0. Start
1. Lies eine Zahl N ein
2. Wenn N eine positive ganze Zahl ist, gehe zu 3, sonst gehe zu 7
3. Setze I gleich 1 und SUMME gleich 0
4. Solange I kleiner oder gleich N gilt,
 addiere I zu SUMME und erhöhe I um 1
6. Schreibe "Summe=" SUMME und gehe zu 8
7. Schreibe "Unzulässige Eingabe"
8. Ende

In der einfachsten Form eines Algorithmus (z.B. im ersten Beispiel "Münzfernsprecher") werden alle Anweisungen Schritt für Schritt nacheinander abgearbeitet, d.h. die "physikalische" (statische) Reihenfolge der Anweisungen im Algorithmus entspricht vollständig der "logischen" (dynamischen) Reihenfolge bei der Ausführung.

Im letzten Beispiel wird die Ausführungsreihenfolge allerdings durch **Steuerungsmechanismen** verändert. So wird z.B. in Anweisung 4 eine Anweisung wiederholt ausgeführt, wobei vor jeder Ausführung eine **Bedingung** überprüft wird. Das Ergebnis der Prüfung verändert sich während der Ausführung, wenn der Wert von I den Wert von N erreicht hat.

Im einzelnen wird die **Reihenfolge** der Abarbeitung im letzten Beispiel durch folgende Steuerungsmechanismen bestimmt:

- sequentielle Durchführung von Anweisungen(0,1,3,7,8)
- Test einer Bedingung und anschließende Verzweigung (2)
- wiederholte Durchführung von Anweisungen (4)
- Sprung zu einer anderen Anweisung (6)

3.5. Bausteine eines Algorithmus

Die Anweisungen eines Algorithmus setzen sich aus Verben (**Operationen**) und Substantiven (**Objekten**) zusammen. Sie manipulieren den Zustand eines Objektes durch Operationen bzw. sie verknüpfen Objekte durch Operationen, um neue Objekte zu erzeugen.

Ein Objekt, dessen Wert zu Beginn der Ausführung bekannt ist und sich während der Ausführung des Algorithmus **nicht** ändert, wird als konstantes Objekt oder kurz als **Konstante** bezeichnet. Ein Objekt mit veränderbarem Wert wird als variables Objekt oder kurz als **Variable** bezeichnet.

Im letzten Beispiel gibt es u.a. die Operationen "Lesen", "Addieren" und "Schreiben", die auf den Objekten (genauer Variablen) "N", "I" und "SUMME" operieren.

Ein Algorithmus wird immer aus folgenden elementaren Bausteinen (Typen von Anweisungen) aufgebaut, die beliebig ineinander verschachtelt werden können:

- **Aneinanderreihung**
- **Verzweigung**
- **Wiederholung**
- **Sprung**

Bei einer **Aneinanderreihung** (andere Bezeichnungen: Sequenz, Folge, Block) werden Anweisungen nacheinander ausgeführt. Sprachlich wird eine Aneinanderreihung durch folgende Form ausgedrückt:

1. <Anweisung 1>
 2. <Anweisung 2>
 3. <Anweisung 3>
- ...

Bei einer **Verzweigung** (Auswahl, Entscheidung, Selektion) in einem Algorithmus wird der nächste auszuführende Schritt abhängig von einer Bedingung gemacht. Sprachlich wird eine Verzweigung durch folgende Form ausgedrückt:

Wenn <Bedingung> gilt,
 dann führe <Anweisung 1> aus,
 sonst führe <Anweisung 2> aus.

Bei einer **Wiederholung** (Schleife, Iteration) wird eine Anweisung solange ausgeführt, bis eine Abbruchbedingung erfüllt ist. Sprachlich wird eine Wiederholung durch folgende Form ausgedrückt:

Solange <Bedingung > gilt,
 führe <Anweisung 1> aus.

Bei einem **Sprung** wird der Algorithmus mit einer Anweisung fortgesetzt, deren Nummer als Sprungziel angegeben ist:

Gehe zur Anweisung <Nummer>

3.6. Darstellungsformen




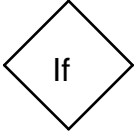


Die Darstellung eines Algorithmus kann, wie bisher beschrieben, textuell oder auch grafisch erfolgen.

In der Literatur werden häufig folgende Darstellungsformen verwendet:

- Natürliche Sprache
- Pseudo-Code
- Flußdiagramm
- Struktogramm

3.7. Beispiel 3 - Flußdiagramm

In einem **Flußdiagramm** (manchmal auch als Programm-Ablauf-Plan, PAP, bezeichnet) werden Symbole zur grafischen Darstellung eines Anweisungstyps verwendet. Der Weg zur nächsten Anweisung im Flußdiagramm führt in der Regel nach unten und wird sonst durch einen Pfeil symbolisiert. Die folgende Tabelle zeigt die Typen von Anweisungen und die entsprechenden grafischen Symbole:

START (Beginn des Algorithmus)	
READ (Einlesen von Daten)	
STATEMENT (Ausführen einer Anweisung)	
TEST (Testen einer Bedingung und anschließende Verzweigen)	
PRINT (Ausgeben von Daten)	
STOP (Ende des Algorithmus)	

Beachten Sie, daß der Anweisungstyp "Wiederholung" durch Kombination elementarer Symbole erzeugt wird und der Anweisungstyp "Sprung" durch einen Pfeil symbolisiert wird.

3.8. Beispiel 4 - Pseudo-Code

Im **Pseudo-Code** werden die Anweisungen in einer an die englische Sprache angelehnten Form beschrieben. Die Darstellung im Pseudo-Code ist nicht normiert und läßt Ihnen deshalb gewisse Freiheiten. Der Algorithmus "Berechnen der Summe der Zahlen von 1 bis N" kann im Pseudo-Code z.B. folgendermaßen dargestellt werden:

```

START
READ N
I:=1, SUM:=0
IF ( I <= N) THEN
  SUM:= SUM + I, I:= I+1
END IF
PRINT SUM
STOP

```

(-> Übung 1)

4. Wie führt ein Computersystem ein Programm aus?

In diesem Kapitel lernen Sie die prinzipielle Arbeitsweise eines Computersystems und die Transformation von einem Quellprogramm in ein Zielprogramm (ausführbares Programm) kennen.

4.1. Arbeitsweise eines Computersystems

Ein typisches Computersystem mit von Neumann-Architektur setzt sich (vereinfacht) aus folgenden Einheiten zusammen:

- Prozessor (inklusive Register und Arithmetik-Einheit)
- interner Speicher (ROM, RAM)
- externer Speicher (Festplatte, Diskette, Magnetband)
- Bildschirm/Drucker
- Tastatur/Maus

Der **Prozessor** stellt die zentrale Instanz eines Computers dar. Er nutzt die ihm zur Verfügung stehenden Betriebsmittel, um ein oder mehrere Programme auszuführen.

Er verarbeitet ein Zielprogramm (ausführbares Programm), indem er es vom externen Speicher in den internen Speicher lädt und dann die im Programm enthaltenen Maschinenbefehle Schritt für Schritt ausführt. Als "Schmierzettel" für Berechnungen dienen ihm Register, mit deren Inhalten er arithmetische Operationen durchführen kann.

Die Addition zweier Zahlen A und B zu einer Zahl C wird z.B. in einem FORTRAN Quellprogramm durch folgende Anweisung repräsentiert:

C=A+B

Die Folge der vom FORTRAN Übersetzer (Compiler) erzeugten Maschinenbefehle, die dieser Anweisung entsprechen, lautet:

Load A into Register 1	A -> R1
Load B into Register 2	B -> R2
Add Register 1 and 2 into Register 3	R3 <- R1 + R2
Save C from Register 3	C <- R3

Sie erkennen an diesem Beispiel, daß Anweisungen in einer Programmiersprache kürzer und prägnanter sind als die Folge der zugehörigen Maschinenbefehle.

Ein Prozessor kann u.a. folgende Typen von Maschinenbefehlen ausführen:

LOAD	lädt den Inhalt einer Speicherzelle in ein Register
STORE	speichert den Inhalt eines Registers in eine Speicherzelle
ADD	addiert den Inhalt von zwei Registern
MULTIPLY	multipliziert den Inhalt von zwei Registern
DIVIDE	dividiert den Inhalt von zwei Registern
TEST	testet, ob der Inhalt eines Registers größer als Null ist
JUMP	setzt die Programmausführung an anderer Stelle fort

4.2. Quell- und Zielprogramm

Die Darstellung einer Lösung in einem Algorithmus und in einem Quellprogramm einer höheren Programmiersprache weisen große Ähnlichkeiten auf. Der Name FORTRAN leitet sich z.B. aus **FORmula TRANslation** ab und verdeutlicht, daß FORTRAN zur einfachen Darstellung von mathematischen Formeln in einer Programmiersprache dienen soll.

4.4. Geschichte von FORTRAN

Folgende Daten charakterisieren die Entwicklung von FORTRAN:

- 1957 entwickelt **John Backus** von der Firma IBM den ersten FORTRAN Compiler. Sein Ziel ist, die nur für Experten verständliche Maschinen- bzw. Assemblersprache durch eine leicht erlernbare und lesbare Sprache zu ersetzen.
- 1964 existieren bereits 43 unterschiedliche "**Dialekte**" auf 16 Systemen, so daß die Notwendigkeit zur Normierung der Sprache akut wird.
- 1966 definiert ANSI die Norm für **FORTRAN 66** (auch als FORTRAN IV bezeichnet).
- 1978 wird FORTRAN 66 durch **FORTRAN 77** abgelöst. FORTRAN 77 beseitigt Schwächen in der ursprünglichen Definition.
- 1992 wird **Fortran 90** definiert.

5. Überblick über die FORTRAN Sprachelemente

In diesem Kapitel erhalten Sie einen einführenden Überblick über die in FORTRAN verfügbaren Sprachelemente. Die Sprachelemente werden in den folgenden Kapiteln systematisch behandelt.

5.1. Vergleich mit einer natürlichen Sprache

Die Sprachelemente der Programmiersprache FORTRAN lassen sich den Sprach- und Gliederungselementen einer natürlichen Sprache in etwa folgendermaßen zuordnen:

Natürliche Sprache	FORTRAN
Zeichen	Zeichen
Wort	Wort
Satzteil	Ausdruck
Satz	Anweisung
Kapitel	Programm-Einheit
Buch	Programm

5.2. Schreibweise, Syntax und Semantik

Wie in einer natürlichen Sprache wird in FORTRAN zwischen der Schreibweise einzelner Wörter und der Grammatik (Syntax) und der Bedeutung (Semantik) von Sätzen unterschieden.

Der Unterschied zwischen orthographisch, syntaktisch und semantisch falschen Sätzen und entsprechenden Fehlern soll an einem Beispiel aus einer natürlichen Sprache illustriert werden:

Der Apfel fellt vom Baum.	-> orthographisch falsch
Der Baum fällt Apfel vom.	-> grammatikalisch falsch
Der Baum fällt von Apfel.	-> semantisch/logisch falsch
Der Apfel fällt vom Baum.	-> ok

Auf die Sprache FORTRAN übertragen existieren folgende Klassen von Fehlern:

- lexikalische Fehler
- syntaktische Fehler
- semantische oder Laufzeit-Fehler
- logische Fehler

Wenn Sie in der Sprache FORTRAN ein Quellprogramm schreiben, prüft der FORTRAN Übersetzer (Compiler) vor der Übersetzung die Schreibweise und Syntax. Er ist nicht (oder nur sehr beschränkt) in der Lage, semantische und logische Fehler zu entdecken.

Insbesondere die Programm-Logik liegt vollständig in Ihrer Verantwortung als Programmierer, und in der Regel bereiten auch die logischen Fehler die größten Probleme bei der Fehlersuche.

5.3. Zeichen

Ein **Zeichen** ist die kleinste lexikalische Einheit, die vom FORTRAN Sprachübersetzer (Compiler) bearbeitet wird.

Der Zeichensatz von FORTRAN besteht aus folgenden Zeichen:

- **Buchstaben und Ziffern** A-Z 0-9
- **Sonderzeichen** = + - * / () , . \$ ' : "
- **zusätzliche Zeichen** (hier: IBM VS FORTRAN): a b ... z _ !

Beachten Sie,

daß Sie alle anderen Zeichen wie z.B. [, β, § und ü nur in Zeichenketten oder in Kommentaren verwenden können.

daß einige FORTRAN Compiler weitere Zeichen zulassen.

5.4. Wörter

Ein **Wort** ist die kleinste syntaktische Einheit (Bedeutungsträger, Baustein) für den FORTRAN Sprachübersetzer (Compiler). Ein Wort setzt sich aus den in FORTRAN gültigen Zeichen zusammen.

In FORTRAN gibt es folgende Klassen von Wörtern:

- **Literale Konstanten** (Zahlen, Zeichenketten und Wahrheitswerte)
- **Namen** (symbolische Konstanten, Variablen, ...)
- **Operatoren** (+, -, *, **, /, .AND., .OR., ...)
- **FORTRAN Schlüsselwörter** (PROGRAM, END, READ, DO, ...)

Beachten Sie,

daß Namen immer mit einem Buchstaben beginnen müssen, damit ein Name sich eindeutig von einer Zahl unterscheidet.

daß Sie Wörter zur Verbesserung der Lesbarkeit durch Leerzeichen oder Tabulatorzeichen voneinander trennen, obwohl es die Sprachdefinition von FORTRAN nicht verlangt.

5.5. Ausdrücke

Ein **Ausdruck** entsteht durch Verknüpfung von Variablen und Konstanten durch Operatoren, z.B. ist $(4*3)$ ein Ausdruck, bei dem 2 literale Konstanten durch den Multiplikationsoperator verknüpft werden.

In FORTRAN gibt es folgende Klassen von Ausdrücken:

- **arithmetische Ausdrücke** (Auswertung ergibt Zahl)
- **logische Ausdrücke** (Auswertung ergibt Wahrheitswert)
- **Zeichenketten-Ausdrücke** (Auswertung ergibt Zeichenkette)

5.6. Anweisungen

Eine **FORTRAN Anweisung** entspricht einer Anweisung eines Algorithmus. Sie wird durch ein FORTRAN Schlüsselwort eingeleitet (Ausnahme: Zuweisung), das die auszuführende Operation in englischer Sprache umschreibt und enthält Objekte (Konstanten und Variablen), die manipuliert werden.

In FORTRAN gibt es folgende Klassen von Anweisungen:

- **vereinbarende Anweisungen**
- **ausführbare Anweisungen**

Sie geben dem FORTRAN Sprachübersetzer (Compiler) mit Hilfe von vereinbarenden Anweisungen vor der Programmausführung z.B. Informationen darüber, wieviel Speicherplatz für Konstanten und Variablen reserviert werden soll und in welcher Form Eingaben und Ausgaben des Programms erfolgen.

Die ausführbaren Anweisungen werden in Maschinenbefehle übersetzt und nach dem Start des Programms in der vorgegebenen Reihenfolge vom Prozessor abgearbeitet.

5.7. Programm-Einheit

Eine **Programm-Einheit** realisiert einen Teilaspekt eines Algorithmus wie z.B. die Interaktion mit dem Benutzer oder die Berechnung einer komplexen Funktion.

Sie fassen mehrere, logisch zusammengehörende Anweisungen in einer Programm-Einheit zusammen, um Ihren Algorithmus zu strukturieren.

In FORTRAN gibt es folgende Klassen von Programm-Einheiten:

- **Haupt-Programm** (**PROGRAM - ... - END**)
- **Funktion** (**FUNCTION -... - END**)
- **Unter-Programm** (**SUBROUTINE - ... - END**)
- **Datenblock** (**BLOCK DATA - ... - END**)

5.8. Programm

Ein **Haupt-Programm**, zusammen mit den von ihm aufgerufenen Programm-Einheiten, wird kurz als Programm bezeichnet. Ein Programm enthält somit immer **genau eine** Programm-Einheit "Haupt-Programm" und ggfs. weitere Programm-Einheiten aus anderen Klassen.

Im weiteren Sinne umfaßt ein Programm neben den aufgerufenen Programmeinheiten auch die zugehörigen Dateien mit Eingabedaten.

6. Elementare Sprachelemente

In diesem Kapitel lernen Sie elementare FORTRAN Sprachelemente kennen, mit denen Sie einfache FORTRAN Programme realisieren können.

6.1. Literale Konstanten

Eine **literale Konstante** entspricht einem unveränderbaren Objekt eines Algorithmus.

Eine literale Konstante repräsentiert einen Wert, den Sie zu Beginn der Programmausführung festgelegt haben und während der Programmausführung nicht mehr verändern können (**konstanter Wert**).

Literale Konstanten gehören zu einem der folgenden elementaren FORTRAN Datentypen:

Bezeichnung	FORTRAN Datentyp	Beispiel
ganze Zahl	INTEGER	100
reelle Zahl	REAL	2.5E10
reelle Zahl (hohe Genauigkeit)	DOUBLE PRECISION	2.888D3
komplexe Zahl	COMPLEX	(3.0,1)
Zeichen	CHARACTER	'a'
Zeichenkette (maximal n Zeichen)	CHARACTER*n	'abc'
Wahrheitswert	LOGICAL	.TRUE.

Literale Konstanten sind demnach charakterisiert durch

- **Datentyp**
- **Wert (unveränderbar)**

6.1.1. INTEGER

Eine literale Konstante vom Datentyp INTEGER repräsentiert eine ganze Zahl. Beispiele für literale Konstanten vom Datentyp INTEGER, kurz INTEGER Konstanten, sind:

0
300
-50

integer constant := [- | +]nnnnn

n=0-9

Folgende Literale sind als INTEGER Konstanten nicht zulässig:

3.0 -> Punkt unzulässig
20000000000 -> Wertebereich überschritten (systemabhängig!)
'2' -> Apostroph unzulässig

6.1.2. REAL

Eine literale Konstante vom Datentyp REAL repräsentiert eine reelle Zahl mit endlicher Stellenzahl:

3.0	repräsentiert 3.0
.0	repräsentiert 0.0
3.E-5	repräsentiert $3,0 \times 10^{-5}$
4.3E+10	repräsentiert $4,3 \times 10^{10}$

real constant := [-|+]vvvvv.nnnnn[E[+|-]eee]

v,n,e=0-9

Folgende Literale sind nicht zulässig:

3,0	-> Komma unzulässig
7E5	-> Punkt zwingend notwendig
4.E-0.5	-> reeller Exponent (hier: eee=-0.5) unzulässig

6.1.3. DOUBLE PRECISION

Eine literale Konstante vom Datentyp DOUBLE PRECISION repräsentiert eine reelle Zahl mit höherer interner Genauigkeit (mehr Nachkommastellen) im Vergleich zum Datentyp REAL:

2.01234567890123D-3	repräsentiert 0.0020134567890123
3.14159265358979D0	repräsentiert 3.14159265358979 (Pi)

double precision constant := [-|+]vvvvv.nnnnnD[+|-]eee

v,n,e=0-9

Folgende Literale sind nicht zulässig:

3,0D1	-> Komma unzulässig
7.0E5	-> E definiert REAL

6.1.4. COMPLEX

Eine literale Konstante vom Datentyp COMPLEX repräsentiert eine komplexe Zahl (einen Punkt in der komplexen Ebene):

(3.0,4.0)	repräsentiert $(3.0 + I 4.0)$
(.5E-2,0.2E10)	repräsentiert $(0.5 \times 10^{-2} + I 0.2 \times 10^{10})$

complex constant := (real constant, real constant)

Folgende Literale sind nicht zulässig:

3.0+I*2.0	-> I Notation unzulässig
-----------	--------------------------

(3.0 4.0) -> trennendes Komma fehlt

6.1.5. CHARACTER

Eine Konstante vom Datentyp CHARACTER repräsentiert ein einzelnes Zeichen:

'a' repräsentiert das Zeichen a, LEN('a')=1

character constant := 'c'

c=a-z | A-Z | 0-9 | ...

Folgende Literale sind nicht zulässig:

"a" -> Begrenzer unzulässig (systemabhängig)
 b -> Begrenzer fehlen
 'ab' -> nur ein Zeichen zulässig (LEN('ab') = 2)

Beachten Sie,

daß die Begrenzer '...' **nicht** zur Konstanten c gehören, sondern lediglich Anfang und Ende des Zeichens c kennzeichnen

6.1.6. CHARACTER*n

Eine literale Konstante vom Datentyp CHARACTER*n repräsentiert eine Zeichenkette der Länge n, z.B. sind im Datentyp CHARACTER*5 Zeichenketten der Länge 5 darstellbar:

'Hello World' repräsentiert eine Zeichenkette Hello World der Länge 11

character*n constant := 'cccc...c'

c=a-z | A-Z | 0-9 | ...

Folgende Literale sind nicht zulässig:

"abc" -> Begrenzer unzulässig (systemabhängig)
 'ab -> rechter Begrenzer fehlt
 " -> Länge 0 unzulässig

Beachten Sie,

daß die Begrenzer ' bzw. " **nicht** zur Zeichenkette gehören, sondern lediglich Anfang und Ende kennzeichnen.

daß Sie in einer Zeichenkette ein Apostroph als "normales" Zeichen verwenden können, indem Sie es verdoppeln.

daß die Begrenzer nicht bei der Berechnung der Länge mitgezählt werden.

6.1.7. LOGICAL

Eine literale Konstante vom Datentyp LOGICAL repräsentiert einen von zwei möglichen Wahrheitwerten:

.TRUE. repräsentiert den Wahrheitswert 'Wahr'

`.FALSE.` repräsentiert den Wahrheitswert 'Falsch'

logical constant := `.TRUE.`
logical constant := `.FALSE.`

Alle anderen Schreibweisen sind ungültig.

6.2. Symbolische Konstanten

Eine **symbolische Konstante** ist ein Name für eine literale Konstante.

Sie können eine literale Konstante in einem Programm durch einen (möglichst aussagekräftigen) symbolischen Namen bezeichnen. Ein symbolischer Name für eine literale Konstante muß wie alle Namen mit einem Buchstaben beginnen und darf nur die für FORTRAN zulässigen Zeichen enthalten.

Zweckmäßigerweise kennzeichnen Sie alle symbolischen Konstanten eines Datentyps mit einem einheitlichen Anfangsbuchstaben, z.B. `i` für den Datentyp `INTEGER` und `r` für `REAL`.

Eine symbolische Konstante repräsentiert eine literale Konstante und gehört deshalb wie diese zu einem der folgenden elementaren Datentypen:

Bezeichnung	FORTRAN Datentyp	Beispiel
ganze Zahl	<code>INTEGER</code>	<code>(iMAX=100)</code>
reelle Zahl	<code>REAL</code>	<code>(rCONST1=2.5E10)</code>
reelle Zahl (hohe Genauigkeit)	<code>DOUBLE PRECISION</code>	<code>(dC=2.888D3)</code>
komplexe Zahl	<code>COMPLEX</code>	<code>(cSTART=(1,1))</code>
Zeichen	<code>CHARACTER</code>	<code>(chLETTERA='a')</code>
Zeichenkette (maximal n Zeichen)	<code>CHARACTER*n</code>	<code>(ch3ABC='abc')</code>
Wahrheitswert	<code>LOGICAL</code>	<code>(ITRUE=.TRUE.)</code>

Symbolische Konstanten sind demnach durch folgende Eigenschaften charakterisiert:

- **Symbolischer Name**
- **Datentyp**
- **Wert (unveränderbar)**

Sie können z.B. die reelle Konstante `rPI` und `rEULER` und die Zeichenketten-Konstanten `chMSG1` folgendermaßen definieren und mit Werten versehen:

```
REAL          rPI
PARAMETER    (rPI = 3.1415926)

CHARACTER*80 chMSG1
PARAMETER    (chMSG1='Hello World!')
```

Die Syntax für die **Typdefinition und Wertzuweisung für eine symbolische Konstante** `name1` lautet:

type name1 [, ...]
PARAMETER (name1 = value1 [, ...])

Sie sollten aus folgenden Gründen symbolische Konstanten anstelle von literalen Konstanten verwenden:

- Symbolische Konstanten sind vorteilhaft, um ein Programm "änderungsfreundlich" zu gestalten. Sie brauchen nur an einer Stelle zu ändern, nämlich in der PARAMETER Anweisung.
- Symbolische Konstanten sind in der Regel aussagekräftiger als reine Zahlen, z.B. ist rMWST (für Mehrwertsteuer) besser zu interpretieren als 0.15.

6.3. Variablen

Eine **Variable** entspricht einem veränderbaren Objekt eines Algorithmus.

Eine Variable repräsentiert eine Speicherstelle, deren Wert Sie während der Programmausführung verändern können (**variieren** können). Eine Variable wird durch einen (möglichst aussagekräftigen) symbolischen Namen bezeichnet. Der symbolische Name muß mit einem Buchstaben beginnen und darf nur die für FORTRAN zulässigen Zeichen enthalten.

Variablen gehören wie Konstanten zu einem elementaren Datentyp:

Bezeichnung	FORTRAN Datentyp	Beispiel
ganze Zahl	INTEGER	iMaxV=100
reelle Zahl	REAL	r1=2.8E10
reelle Zahl (höhere Gen.)	DOUBLE PRECISION	d1=2.888D3
komplexe Zahl	COMPLEX	c1=(1.0,1.0)
Zeichen	CHARACTER	chLetter='a'
Zeichenkette	CHARACTER*n	chWord='abc'
Wahrheitswert	LOGICAL	b1=.TRUE.

Variablen sind demnach durch folgende Eigenschaften charakterisiert:

- **Symbolischer Name**
- **Datentyp**
- **Wert (veränderbar)**

Sie definieren den Datentyp einer Variablen durch eine Typvereinbarung. Sie können und sollten ihr gleichzeitig einen Anfangswert zuweisen (systemabhängig!). Sie können z.B. Typ und Anfangswert der Variablen rCircleArea und rCircleRadius folgendermaßen vereinbaren:

```
REAL rCircleArea /0.0/, rCircleRadius /0.0/
```

Die Syntax der **Typvereinbarung und Anfangswertzuweisung für eine Variable** var1 lautet:

```
type var1 [/value1/] [, var2 /value2/ ...]
```

Achtung:

Einige Compiler verwenden die veraltete **DATA Anweisung** für eine Anfangswertzuweisung.

Falls für eine im Programm benutzte Variable **keine** Typvereinbarung existiert, erfolgt eine **implizite Typvereinbarung** anhand des Anfangsbuchstabens:

I-N: INTEGER, alle anderen: REAL

6.4. Arithmetische Ausdrücke

Ein **arithmetischer Ausdruck** entsteht durch Verknüpfung von Variablen und Konstanten.

Sie können arithmetische Konstanten und Variablen mit Hilfe von arithmetischen Operatoren zu komplizierten arithmetischen Ausdrücken zusammensetzen. Sie können z.B. folgende einfachen Ausdrücke bilden:

```
3*(4-8)
5**2
3.0/4.0*rPI
```

Die Syntax für die **Bildung eines Ausdrucks** aus zwei Ausdrücken expr1 und expr2 mit dem verknüpfenden Operator op lautet:

expr3 := expr1 op expr2

In der Sprache FORTRAN sind folgende arithmetische Operatoren mit fallender Priorität definiert:

Operator	Bedeutung	Priorität
**	Potenzieren	0
-/+	Vorzeichen	1
*	Multiplizieren	2
/	Dividieren	2
+	Addieren	3
-	Subtrahieren	3

Beachten Sie folgende Regeln:

- In einem Ausdruck werden Operatoren gleicher Priorität **von links nach rechts** ausgewertet. Eine Ausnahme stellt das **Potenzieren** dar, das **von rechts nach links** ausgewertet wird.
- Benutzen Sie **Klammern**, falls Sie die Auswertungsreihe verändern wollen
- Bei Ausdrücken, die unterschiedliche Datentypen enthalten, wird vom FORTRAN Sprachübersetzer (Compiler) eine **automatische Umwandlung** vom "schwächeren zum stärkeren Datentyp" durchgeführt.

6.5. Beispiel 5 - Auswertungsreihenfolge für einen Ausdruck

Aufgrund der Prioritäten von Operatoren wird der Ausdruck $(4**2+6/3)$ folgendermaßen ausgewertet:

$(4**2)+(6/3)$	(** besitzt höchste Priorität.)
$16+(6/3)$	(Nun besitzt / höchste Priorität.)
$16+2$	(Addition besitzt niedrigste Priorität.)
18	

6.6. Beispiel 6 - Automatische Umwandlung

Aufgrund der Regeln für die automatische Umwandlung wird der Quotient zweier INTEGER Ausdrücke auf einen INTEGER Ausdruck abgebildet. Der resultierende Ausdruck ist also in der Regel gerundet (im folgenden bezeichnet INT das Abschneiden der Dezimalstellen):

```
5 / 2
= INT(2.5)
= 2
```

Bei der Addition von einem REAL Ausdruck und einem INTEGER Ausdruck ist der resultierende Ausdruck vom stärkeren Datentyp, also vom Datentyp REAL:

```
3 + 1.5      (unterschiedliche Typen)
3.0 + 1.5    (automatische Umwandlung von 3 zu 3.0)
4.5          (Ergebnis vom Typ REAL)
```

6.7. Arithmetische Zuweisungen

Eine **Zuweisung** verändert den Wert einer Variablen.

Eine Variable bezeichnet eine Speicherstelle, deren Inhalt zunächst durch eine Anfangswertzuweisung definiert und dann während des Programmes durch weitere Zuweisungen verändert wird. Sie können z.B. die Werte der arithmetischen Variablen rCircleArea und iSum durch folgende Zuweisungen verändern:

```
rCircleArea = rPI * rCircleRadius**2
iSum = 17
```

Die Syntax der **Zuweisung eines Ausdrucks** **exp** **an eine Variable** **var** lautet:

```
var = expr
```

6.8. Beispiel 7 - Kreisberechnung

Wir betrachten ein erstes vollständiges Programm SAMP1 zur Berechnung der Kreisfläche. Die Kreiskonstante Pi wird in einer PARAMETER Anweisung als symbolische Konstante rPI eingeführt.

Sie werden die bereits hier verwendeten FORTRAN Schlüsselwörter PROGRAM, READ, PRINT und END und die Einrückung des Quelltextes um 6 Zeichen in Kürze kennenlernen.

```
*234567890123456789012345678901234567890...01234567890
   1           2           3           4       7           8

PROGRAM SAMP1
REAL rPI, rCircleRadius /0.0/, rCircleArea /0.0/
PARAMETER (rPI=3.141592)
READ *, rCircleRadius
rCircleArea = rPI * (rCircleRadius**2)
PRINT *, rCircleArea
END
```

6.9. Beispiel 8 - Variablen in Zuweisungen

Das folgende Programm SAMP2 demonstriert, wie die Variable iSum in einer Zuweisung ausgewertet und gleichzeitig verändert werden kann:

```
PROGRAM SAMP2
INTEGER iSum /0/, i1 /1/
iSum = iSum + i1
PRINT *, i1, iSum
END
```

Beachten Sie folgende Unterschiede:

Wenn eine Variable **links** vom Gleichheitszeichen steht, ist ihr **Name** (entspricht der **Adresse** einer Speicherstelle) gemeint. Die Zuweisung (iSum = 1) bewirkt z.B., daß die Adresse der Speicherstelle von iSum bestimmt und der Inhalt der Speicherstelle auf 1 gesetzt wird.

Wenn eine Variable **rechts** vom Gleichheitszeichen steht, ist ihr **aktueller Wert** (entspricht dem aktuellen **Inhalt** einer Speicherstelle) gemeint. Bei der Auswertung einer rechten Seite, in der eine Variable auftaucht, wird deren aktueller Wert eingesetzt.

Wenn eine Variable sowohl **links** als auch **rechts** in einer Zuweisung auftaucht wie z.B. (iSum = iSum + i1), ergibt sich der neue Inhalt der Speicherstelle von iSum aus der Addition des letzten Inhalts der Speicherstelle von iSum erhöht um den Wert von i1. Die Speicherstelle iSum wird dabei **zuerst** (rechts) gelesen und **danach** (links) verändert.

6.10. Beispiel 9 - Genauigkeitsverlust

Bei der Zuweisung eines typstärkeren Ausdrucks an eine typschwächere Variable wird der zugewiesene Wert an den schwächeren Typ angepaßt und verliert in der Regel an Genauigkeit:

```
INTEGER i1
i1 = 1.5           ! i1 = 1, Dezimalstelle geht verloren

REAL r1
r1 = (1.0, 1.0)   ! r1 = 1.0, Imaginärteil geht verloren
```

6.11. Listengesteuerte Ein- und Ausgabe

Für die Operationen "Lies" und "Schreibe" in Algorithmen gibt es einfache und komplizierte Entsprechungen im Sprachumfang von FORTRAN.

Im folgenden lernen Sie zunächst die Anweisungen READ *, ... und PRINT *, ... kennen, die Werte für Variablen von der Standardeingabe * (**Tastatur**) einlesen bzw. Werte von Variablen, Konstanten oder Ausdrücken in die Standardausgabe * (**Bildschirm**) ausgeben.

Das Zeichen * bedeutet in diesem Zusammenhang, daß eine Standardvorgabe für die Ein- bzw. Ausgabeeinheit benutzt werden soll. Die Datentypen werden automatisch ermittelt. Diese Form der Ein-/Ausgabe wird als listengesteuert bezeichnet.

Sie können z.B. folgendermaßen Werte für die Variablen iCount und rCircleRadius einlesen und ausgeben:

```
PRINT *, 'Geben Sie einen Wert für iCount ein (ganzzahlig):'
READ *, iCount
PRINT *, 'Geben Sie einen Wert für rCircleRadius ein (reellwertig):'
READ *, rCircleRadius

PRINT *, 'Der Wert für iCount lautet: ', iCount, '.'
PRINT *, 'Der Wert für rCircleRadius lautet: ', rCircleRadius, '.'
```

Die Syntax für das **listengesteuerte Einlesen** von Variablen und das **listengesteuerte Ausgeben** von Ausdrücken lautet:

```
READ *, var1 [,var2 ...]
PRINT *, expr1 [, expr2 ...]
```

6.12. Beginn und Ende eines Programmes

Die Syntax von FORTRAN verlangt, daß Sie den Beginn und das Ende einer Programm-Einheit besonders kennzeichnen. Sie müssen für ein Haupt-Programm die FORTRAN Schlüsselwörter **PROGRAM** und **END** verwenden.

Die Syntax für die **Kennzeichnung von Beginn und Ende** eines Haupt-Programmes pgm lautet:

```
PROGRAM [pgm]
...
END
```

6.13. Beispiel 10 - Kreisberechnung

Sie können nun das Programm SAMP3 zur Kreisberechnung folgendermaßen benutzerfreundlicher gestalten. Die **IMPLICIT** Anweisung in der Form **IMPLICIT NONE** bewirkt, daß der FORTRAN Sprachübersetzer (Compiler) keine implizite Typvereinbarung zuläßt:

```
PROGRAM SAMP3
IMPLICIT NONE
REAL rPI, rCircleRadius /0.0/,
+      rCircleArea /0.0/, rCircleCirc /0.0/
PARAMETER (rPI=3.1415926)
PRINT *, 'Geben Sie den Kreisradius ein:'
READ *, rCircleRadius
rCircleArea = rPI * (rCircleRadius**2)
rCircleCirc = 2 * rPI * rCircleRadius
PRINT *, 'Die Kreisflaeche betraegt: ', rCircleArea
PRINT *, 'Der Kreisumfang betraegt: ', rCircleCirc
END
```

(-> Übung 2)

(-> Übung 3)

6.14. Zusammenfassung

In diesem Kapitel haben Sie folgende elementaren Sprachelemente kennengelernt:

- **arithmetische Ausdrücke**
- **arithmetische Operatoren**
- **arithmetische Zuweisungen**
- **IMPLICIT Anweisungen**
- **listengesteuertes Ein- und Ausgeben**
- **literale Konstanten**
- **PARAMETER Anweisungen**
- **PROGRAM/END Anweisungen**
- **symbolische Konstanten**
- **Typvereinbarungen**
- **Variablen**

Auf Grundlage dieser Sprachelemente können Sie Programme folgender Bauart entwickeln:

Beginn des Programms	PROGRAM pgm
Verbieten von impliziten Typvereinbarungen	IMPLICIT NONE
Typvereinbarungen	REAL rVar1, rVar2, rCONST1
Definitionen von Konstanten	PARAMETER (rCONST1=2.843)
Einlesen von Variablen	READ *, rVar1
Zuweisungen	rVar2 = rCONST1 * rVar1
Ausgeben von Ausdrücken	PRINT *, 'rVar2 = ', rVar2
Ende des Programms	END

Beachten Sie,

daß Sie in FORTRAN Programm Anweisungen immer in einer festgelegten Reihenfolge anordnen müssen (siehe nächste Tabelle).

	PROGRAM	
		PARAMETER
		DATA
	IMPLICIT	
		andere Vereinbarungen (z.B. REAL ...)
		ausführbare Anweisungen (z.B. READ *, ...)
END		

7. Festes Eingabeformat und Kommentare

In diesem (recht kurzen) Kapitel lernen Sie die Regeln für das feste Eingabeformat kennen, die Sie in den Programmierübungen bereits eingehalten haben. Ferner lernen Sie die Möglichkeit kennen, Programme durch Kommentare zu dokumentieren.

7.1. Aufteilung der verfügbaren Spalten

Sie müssen in FORTRAN aus historischen Gründen (-> Lochkarten) das Quellprogramm in einem festgelegten Format schreiben. Die insgesamt 80 Spalten einer Eingabezeile werden dabei in 4 Bereiche eingeteilt:

```

1234567890123456789012345678901234567890...01234567890
      1           2           3           4       7           8
MMMMM+AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...AAACCCCCCCC

      PROGRAM ...
      REAL ...
    
```

Symbol	Spalten	Bedeutung
M	1-5	maximal 5-ziffrige Sprungmarke (label)
+	6	1-stellige Kennzeichnung für eine Fortsetzungszeile
A	7-72	Anweisungstext
C	72-80	Kommentartext

Beachten Sie,

daß für den **Anweisungstext** die Spalten 7 bis 72 möglich sind.

daß eine Anweisung auf mehreren Zeilen fortgesetzt werden kann, wobei Sie die **Fortsetzung** in Spalte 6 durch ein beliebiges Zeichen (empfohlen: '+' oder '&') kennzeichnen müssen.

daß Sie die restlichen Spalten können für **Sprungmarken** (1-5) bzw. für **Kommentar** (73-80) verwenden oder auch frei lassen können.

daß Sie die **Tabulatoren** in Ihrem Editor möglichst auf Spalte 1, 6 und 7 einstellen sollten.

Beispiel 11 - Quellprogramm mit Fortsetzungszeilen

Im folgenden Beispiel werden Fortsetzungszeilen und Sprungmarken verwendet:

```

*234567890123456789012345678901234567890...01234567890
*      1           2           3           4       7           8

      PROGRAM ...
      REAL rPI,
      &    rCircleArea,
      &    rCircleDiam
      ...
10000 END
    
```

Beachten Sie,

daß die Anweisung aus Zeile 2 in den Zeilen 3 und 4 fortgesetzt wird.

daß die END Anweisung mit der Sprungmarke 10000 versehen ist, so daß ein Sprung zu dieser Anweisung (z.B. mit GOTO 10000) erfolgen kann.

7.2. Kommentare

Jede Leerzeile oder jede Zeile, die in Spalte 1 das Zeichen '*' oder 'C' enthält, wird als reine Kommentarzeile betrachtet.

Es ist empfehlenswert, wichtige oder "trickreiche" Stellen im Programm geeignet zu kommentieren.

Viele Compiler (z.B. auch der VS FORTRAN Compiler) bieten zusätzlich die Möglichkeit, einen Kommentar innerhalb einer Anweisungszeile durch '!' einzuleiten (inline comment). In diesem Fall wird der Text nach '!' bis zum Zeilenende als Kommentar betrachtet.

7.3. Beispiel 12 - Kommentare

Das folgende Programmfragment enthält 5 Kommentarzeilen:

```
*234567890123456789012345678901234567890...01234567890
*      1          2          3          4    7          8
*      Das ist eine Kommentarzeile
C      Das ist auch eine Kommentarzeile
PRINT *, rPI ! Das ist ein Kommentar am Ende einer Zeile
```

Der FORTRAN Übersetzer (Compiler) entfernt sämtliche Kommentare vor der Übersetzung, so daß nur folgende FORTRAN Anweisung zur Übersetzung übrig bleibt:

```
PRINT *, rPI
```

7.4. Zusammenfassung

Sie geben **Quelltext** in folgender Form ein:

Spalte Inhalt

- | | |
|-------|--------------------------------------------------------|
| 1-5 | Sprungmarke (optional) |
| 6 | Fortsetzungskennzeichnung (nur für Fortsetzungszeilen) |
| 7-72 | Anweisungstext |
| 73-80 | Kommentar |

Sie geben **Kommentar** in folgender Form ein:

1. Eine Leerzeile oder eine Zeile, die in Spalte 1 das Zeichen C bzw. * enthält, wird vollständig als Kommentar behandelt.
2. Der gesamte Text nach dem Zeichen ! bis zum Ende der Zeile wird als Kommentar behandelt.

8. Wiederholungen, Verzweigungen und Sprünge

In diesem Kapitel lernen Sie die FORTRAN Sprachelemente zur Realisierung einer Wiederholung und einer Verzweigung kennen. Im Zusammenhang mit der Wiederholungsanweisung lernen Sie Felder kennen und in Zusammenhang mit der Verzweigungsanweisung logische Ausdrücke und Zuweisungen.

8.1. DO Anweisungen (Wiederholungen)

Eine **DO Anweisung** realisiert eine Wiederholung eines Algorithmus.

Sie können mit Hilfe der DO Anweisung eine Anweisung bzw. einen Anweisungsblock mehrfach ausführen. Die DO Anweisung setzt sich aus Schleifenkopf, Schleifenkörper und Schleifenende zusammen:

DO i1=iStart, iMax, iStep	-> Schleifenkopf
<statement>	-> Schleifenkörper
...	
END DO	-> Schleifenende

Sie können z.B. eine PRINT Anweisung folgendermaßen 10-mal ausführen:

```
DO i1=1, 10, 1
    PRINT *, 'Durchlauf Nr.', i1
END DO
```

Die Ausführung dieser DO Anweisung erzeugt folgende Bildschirmausgabe:

```
Durchlauf Nr. 1
Durchlauf Nr. 2
...
Durchlauf Nr. 10
```

Die Syntax für eine **DO Anweisung** (Schleife, Wiederholung) mit der Schleifenvariablen i1 und den Schleifenparametern iStart, iMax und iStep lautet:

```
DO i1=iStart, iMax [, iStep]
    statement(s)
END DO
```

Achtung:

Der FORTRAN Standard verlangt folgende Syntax:

```
DO label i1=iStart, iMax [, iStep]
    ...
label CONTINUE
```

Beachten Sie folgende Regeln:

Die Anzahl N der Schleifendurchläufe steuern Sie durch die 3 Schleifenparameter Startwert, Endwert und Schrittweite (im Beispiel: iStart=1, iMax=10 und iStep=1):

$$N = (\text{Endwert} - \text{Startwert} + \text{Schrittweite}) / \text{Schrittweite}$$

$$= (10 - 1 + 1) / 1 = 10$$

Die Schleifenvariable (im Beispiel i1) wird zu Beginn auf den Startwert gesetzt und dann bei jedem Durchlauf der Schleife um den konstanten Wert Schrittweite vergrößert bzw. verkleinert. Die Schrittweite beträgt standardmäßig 1. Negative Schrittweiten, z.B. -1, sind zulässig; in diesem Fall müssen Sie allerdings den Endwert kleiner als den Startwert wählen, weil sonst die Schleife kein einziges Mal durchlaufen wird. Die Ausführung der DO Anweisung wird beendet, wenn die Schleifenvariable (im Beispiel i1) einen Endwert (im Beispiel 10) erreicht hat.

Die Schleifenvariable (im Beispiel i1) kann innerhalb des Schleifenkörpers und nach Beendigung der Schleife mit ihrem aktuellen Wert gelesen werden.

Sie können die Ausführung eines Schleifendurchlaufs mit einer **CONTINUE** Anweisung abbrechen. Damit starten Sie sofort den nächsten Schleifendurchlauf.

Sie dürfen nicht in eine Schleife hereinspringen und sollten möglichst nicht aus einer Schleife herausspringen.

Im allgemeinen Fall mit **reellwertigen** Schleifenparametern rStart, rMax und rStep wird die Anzahl der Schleifendurchläufe N durch folgende Formel bestimmt:

$$N1 = \text{INT}((r\text{Max} - r\text{Start} + r\text{Step}) / r\text{Step})$$

$$N = \text{MAXIMUM}(N1, 0)$$

Beispiel 13 - Berechnung von 2 hoch N

Das folgende Programm SAMP4 berechnet die Folge der Potenzen 2^N für $N=1, 2, \dots, 15$:

```
PROGRAM SAMP4
  INTEGER i1, iPot, iMAX
  PARAMETER (iMAX=15)
  DO i1=1, iMAX
    iPot = 2** i1
    PRINT *, '2**', i1, ' = ', iPot
  END DO
END
```

8.2. Felder

Ein **Feld** ist ein zusammengesetzter Datentyp, der aus mehreren Variablen des selben Datentyps besteht..

Sie können gleichartige Variablen wie z.B. Meßwerte zur Vermeidung von Schreiarbeit in einem Feld zusammenfassen. Die einzelnen Elemente eines Feldes kennzeichnen Sie durch einen **Index**. Jedes Element eines Feldes können Sie im Programm genauso wie eine Variable verwenden.

Sie können z.B. mit folgender Typvereinbarung ein ganzzahliges Feld iVector mit 30 ganzzahligen Elementen iVector(1), ..., iVector(30) definieren:

```
INTEGER iVector(1:30)
```

Sie können z.B. dem 10. Element des Feldes iVector folgendermaßen den Wert 5 zuweisen:

```
iVector(10) = 5
```

Die Syntax der **Typvereinbarung für ein 1-dimensionales Feld** array lautet:

```
type array(UpperBound)
type array(iLowerBound:iUpperBound)
```

Die Syntax der **Typvereinbarung für ein n-dimensionales Feld** array (n maximal 7) lautet

```
type array(UpperBound1, UpperBound2, ...)
type array(iLowerBound1:iUpperBound1,...)
```

Beachten Sie folgende Regeln:

Sie definieren insgesamt $N=(iUpperBound-iLowerBound+1)$ Feldelemente. Die untere Grenze `iLowerBound` wird standardmäßig auf 1 gesetzt, wenn Sie hierfür keinen Wert in der Typvereinbarung spezifizieren.

Die Auswahl (Indizierung) eines Feldelementes erfolgt über eine ganze Zahl, ggfs. wird gerundet. So wird das dritte Element eines Feldes `rArray` z.B. durch `rArray(3)` oder `rArray(2+1)` oder `rArray(3.5)` ausgewählt.

Sie können bereits bei der Typvereinbarung Anfangswerte zuweisen, wobei das Feld "von links" aufgefüllt wird; im folgenden Beispiel werden die Elemente mit den Indizes 1, 2, 3 und 4 definiert, während das letzte Element undefiniert bleibt:

```
INTEGER iArray(5) /1,1,1,1/
```

Mehrdimensionale Felder werden **spaltenweise** im Speicher abgelegt und dementsprechend auch spaltenweise aufgefüllt ("Der linke Index wandert als schnellster."):

```
INTEGER i1(2,2) /1,2,3,4/
(entspricht: i1(1,1)=1, i1(2,1)=2, i1(2,1)=3, i1(2,2)=4)
```

Beispiel 14 - Berechnung von 2 hoch N (verbessert)

Das folgende Programm SAMP5 berechnet die Folge der Potenzen 2^N für $N=1, 2, \dots, 15$ unter Ausnutzung der bereits berechneten Werte:

```
PROGRAM SAMP5
INTEGER i1, iMAX
PARAMETER (iMAX=15)
INTEGER iPot(iMAX)
iPot(1)=2
PRINT *, '2**', i1, ' = ', iPot(1)
DO i1=2, iMAX
    iPot(i1) = 2 * iPot(i1-1)
    PRINT *, '2**', i1, ' = ', iPot(i1)
END DO
END
```

8.3. Beispiel 15 - Felder in verschachtelten DO Anweisungen

Sie können DO Anweisungen ineinander verschachteln, um z.B. mehrdimensionale Felder zu bearbeiten. Die erste DO Anweisung wird dann als äußere Schleife, die folgende(n) als innere Schleife(n) bezeichnet.

Das folgende Programm SAMP6 berechnet die Transponierte A^T einer Matrix A:

```

PROGRAM SAMP6
INTEGER iNM, i1, i2
PARAMETER (iNM=2)
REAL rMatrix(iNM,iNM) /1,2,3,4/
REAL rMatrixT(iNM,iNM)
INTEGER i1, i2
DO i1=1, iNM
  DO i2=1, iNM
    rMatrixT(i1,i2)=rMatrix(i2,i1)
    PRINT *, 'rMatrix (' ,i1,',',',i2,',')=',rMatrix(i1,i2)
    PRINT *, 'rMatrixT(' ,i1,',',',i2,',')=',rMatrixT(i1,i2)
  END DO ! i2
END DO ! i1
END
    
```

(-> Übung 4)

8.4. Logische Ausdrücke

Ein **logischer Ausdruck** entsteht durch Verknüpfung von logischen Objekten durch logische Operatoren.

Ein logischer Ausdruck entsteht ähnlich wie ein arithmetischer Ausdruck aus der Verknüpfung von logischen Konstanten, logischen Variablen und logischen Operatoren sowie Vergleichen. Sie können z.B. die logischen Variablen bA und bB mit dem Operator `.AND.` verknüpfen und die logische Variable bC mit dem Operator `.NOT.` verneinen:

<code>bA .AND. bB</code>	(Sind sowohl bA als bB wahr?)
<code>.NOT. bC</code>	(Ist die Negation von bC wahr?)

Sie können in FORTRAN folgende logischen Operatoren benutzen:

Operator	Bedeutung	Beispiel
<code>.NOT.</code>	Negation	<code>.NOT. bA</code>
<code>.OR.</code>	Oder-Verknüpfung	<code>bA .OR. bB</code>
<code>.AND.</code>	Und-Verknüpfung	<code>bA .AND. bB</code>
<code>.EQV.</code>	Gleichheit	<code>bA .EQV. bB</code>
<code>.NEQV.</code>	Ungleichheit	<code>bA .NEQV. bB</code>

Die allgemeine Syntax für einen **logischen Ausdruck** `log_expr3`, der aus der Verknüpfung von `log_expr1` und `log_expr2` durch einen logischen Operator `log_op` entsteht, lautet:

`log_expr3 := [.NOT] log_expr1 log_op log_expr2`

8.5. Vergleiche

Ein **Vergleich** ist ein spezieller logischer Ausdruck, bei dem arithmetische Ausdrücke oder Zeichenketten miteinander verglichen werden.

Ein Vergleich liefert den Wahrheitswert 'Wahr', wenn der Vergleich erfüllt ist, wie z.B. in $(5 > 3)$, und den Wahrheitswert 'Falsch' sonst, wie z.B. in $(rPI == 4)$.

Sie können in Vergleichen folgende **Vergleichsoperatoren** verwenden:

Operator	Synonym	Bedeutung
.EQ.	==	expr1 ist gleich expr2
.LE.	<=	expr1 ist kleiner oder gleich expr2
.GE.	>=	expr1 ist größer oder gleich expr2
.LT.	<	expr1 ist kleiner expr2
.GT.	>	expr1 ist größer expr2
.NE.	!=	expr1 ist ungleich expr2

8.6. IF Anweisungen (Verzweigungen)

Eine **IF Anweisung** realisiert eine Verzweigung eines Algorithmus.

Eine einfache IF Anweisung besteht aus einem logischen Ausdruck als Bedingung, die entweder wahr (Bedingung erfüllt) oder falsch (Bedingung nicht erfüllt) sein kann und einem Wahr-Zweig und einem optionalen Nein-Zweig, in die abhängig vom Ergebnis des Tests verzweigt wird:

IF (condition) THEN	-> Kopf der IF Anweisung
<statement1>	-> Wahr-Zweig
ELSE	-> Trennung zwischen den Zweigen
<statement2>	-> Nein-Zweig
END IF	-> Ende der IF Anweisung

Sie können z.B. folgendermaßen den Wert einer Variablen iN daraufhin testen, ob er größer als Null ist oder nicht und entsprechend verzweigen:

```

IF ( iN > 0 ) THEN
    PRINT *, 'iN > 0'
ELSE
    PRINT *, 'iN <= 0'
ENDIF
    
```

Die Syntax für eine **einfache IF Anweisung** mit 1 oder 2 Zweigen lautet:

```

IF ( log_expr ) THEN
    statement1
[ELSE
    statement2]
END IF
    
```

Sie können auch mehrere Bedingungen hintereinander testen:

```

IF ( ( iN > 0 ) .AND. ( iN < 10 ) ) THEN
    PRINT *, 'iN ist größer als Null und iN ist kleiner als 10'
ELSE IF ( iN <= 0 )
    PRINT *, 'iN ist kleiner oder gleich Null'
ELSE
    PRINT *, 'iN ist grösser als 10'
END IF
    
```

Die Syntax für eine **IF Anweisung mit mehreren Zweigen** lautet:

```

IF ( log_expr1 ) THEN
    
```

```

        statement1
ELSE IF (logl_expr2) THEN
        statement2
ELSE IF(...) THEN
        ...
ELSE
        default_statement
END IF

```

Beispiel 16 - Maximum von zwei Zahlen

Das folgende Programm SAMP7 vergleicht zwei Zahlen, die interaktiv eingelesen werden:

```

PROGRAM SAMP7
INTEGER iVar1 /0/, iVar2 /0/
PRINT *, 'Geben Sie zwei ganze Zahlen ein:'
READ iVar1, iVar2
IF ( iVar1 .LT. iVar2) THEN
    PRINT *, iVar1, ' ist kleiner als ', iVar2
ELSE IF( iVar1 .EQ. iVar2) THEN
    PRINT *, iVar1, ' ist gleich ', iVar2
ELSE
    PRINT *, iVar1, ' ist groesser als ', iVar2
ENDIF
END

```

(-> Übung 5)

8.7. Sprungmarken und GOTO Anweisungen

Eine **GOTO Anweisung** realisiert einen unbedingten Sprung.

Sie können die sequentielle Programmausführung durch einen unbedingten Sprung unterbrechen. Hierzu dient die **GOTO Anweisung**, mit der Sie innerhalb einer Programm-Einheit zu einer Sprungmarke (label) springen können, um dort die Ausführung fortzusetzen.

Sie können z.B. die folgende Sprunganweisung verwenden, um die Ausführung an der Anweisung mit der Sprungmarke 100 fortzusetzen:

```

        ...
        GOTO 100
100    ...
        PRINT *, 'Hier ist Marke 100'

```

Die Syntax einer **GOTO Anweisung** lautet:

```

        GOTO label
        ...
label statement

```

Beachten Sie,

daß Sie die GOTO Anweisung nur sparsam verwenden sollten, weil sie die Lesbarkeit eines Programms und das Verständnis der Programmlogik (besonders bei rückwärts gerichteten Sprüngen) erschwert.

8.8. PAUSE Anweisungen und STOP Anweisungen

Sie können die Programmausführung durch eine **PAUSE** Anweisung (nur für interaktive Programme) unterbrechen und durch eine **STOP** Anweisung abbrechen:

```
PAUSE 'Druecken Sie die ENTER Taste, um das Programm fortzusetzen.'
```

```
STOP 'Unzulaessige Eingabe, Programmausfuehrung wird beendet.'
```

Die Syntax für eine **PAUSE** bzw. eine **STOP Anweisung** lautet:

<p>PAUSE 'message text'</p> <p>STOP 'message text'</p>

8.9. Zusammenfassung

Sie haben nun folgende weiteren Sprachelemente kennengelernt:

- **CONTINUE Anweisungen**
- **DO Anweisungen (Wiederholungen)**
- **Felder**
- **GOTO Anweisungen (Sprünge)**
- **IF Anweisungen (Verzweigungen)**
- **logische Ausdrücke**
- **logische Zuweisungen**
- **PAUSE Anweisungen**
- **Sprungmarken**
- **STOP Anweisungen**
- **Vergleiche**

Auf Grundlage dieser Sprachelemente lassen sich Programme folgender Bauart entwickeln:

Beginn des Hauptprogramms	PROGRAM pgm
Verhindern von impliziten Typvereinbarungen	IMPLICIT NONE
Typvereinbarungen von Variablen, Feldern und symbolischen Konstanten	INTEGER iVar1, iVar2, iVar3(30)
Definitionen von symbolischen Konstanten	PARAMETER (rCONST1=2.843)
Einlesen von Variablen und Feldern	READ *, rVar1
arithmetische Zuweisungen	rVar2 = rCONST1 * rVar1
logische Zuweisungen	lVar2 = l1 .AND. l2
DO Anweisungen (Schleifen)	DO i1=1, 100 - ... - END DO
IF Anweisungen (Verzweigungen)	IF (iN > 0) THEN ... END IF
Ausgeben von Ausdrücken	PRINT *, 'rVar2 = ', rVar2
GOTO Anweisungen (Sprünge)	GOTO ...
Ende des Hauptprogramms	END

9. Formelfunktionen, Prozeduren und COMMON Blöcke

In diesem Kapitel lernen Sie Formelfunktionen, Prozeduren und COMMON Blöcke kennen. Mit diesen zusätzlichen Sprachelementen sind Sie in der Lage, Ihre Programme modular aufzubauen und ggfs. einzelne Module auch in mehreren Programmen zu verwenden.

In der Literatur werden Funktionen (Schlüsselwort: **FUNCTION**) und Unterprogramme (Schlüsselwort: **SUBROUTINE**) häufig unter dem gemeinsamen Oberbegriff **Prozedur** oder **Routine** zusammengefaßt. Eine Prozedur (Routine) ist also entweder eine Funktion oder ein Unterprogramm.

9.1. Formelfunktionen

Eine **Formelfunktion** dient zur Abkürzung eines häufig benötigten Ausdrucks.

Eine Formelfunktion verwendet einen oder mehrere Argumente als Formal-Parameter, die beim Aufruf durch Aktual-Parameter ersetzt werden. Sie können z.B. mit den folgenden Formelfunktionen AVG() und SPHERE() den Mittelwert von 2 Zahlen und das Kugelvolumen berechnen.

Sie definieren zunächst die Formelfunktionen in folgender Form:

```
AVG(rVar1, rVar2) = (rVar1 + rVar2) / 2.0
SPHERE(rRadius) = 3.141*(4.0/3.0)*(rRadius**3)
```

Danach können Sie die Formelfunktionen innerhalb des Programms z.B. folgendermaßen aufrufen:

```
rVar3=AVG(3.0,4.0)+AVG(1.0,2.0)      ergibt den Wert 5.0
rVol3=SPHERE(1.0)                  ergibt den Wert (4/3.0)*3.141
```

Die Syntax für die **Typvereinbarung und Definition einer Formelfunktion** func1 lautet:

```
type func1
...
func1(farg1, farg2, ...) = expr (using farg1, farg2, ...)
```

Die Syntax für den **Aufruf einer Formelfunktion** func1 im Programm lautet:

```
... func1(aarg1, aarg2,, ...) ...
```

Beachten Sie,

daß die Definition einer Formelfunktion bereits im Vereinbarungsteil erfolgen muß.

9.2. Beispiel 17 - Berechnung des Kugelvolumens

Das folgende Programm SAMP8 nutzt die Formelfunktion SPHERE() zur Berechnung des Kugelvolumens:

```
PROGRAM SAMP8
INTEGER i1
DOUBLE PRECISION dpi, SPHERE
```

```

PARAMETER (dPI = 3.141592653D0)
SPHERE(dRadius)=dPI*(4.0/3.0)*(dRadius**3)
DO i1=1, 20
  PRINT *, 'Kugelradius: ', (i1/10.0),
&          'Kugelvolumen: ', SPHERE(i1/10.0)
END DO
END
    
```

9.3. Funktionen

Eine **Funktion** realisiert einen Teilschritt eines Algorithmus, bei dem ein Funktionswert berechnet wird.

Sie können neben den einzeiligen Formelfunktionen auch umfangreichere Funktionen verwenden, die Sie allerdings in einer **eigenen** Programm-Einheit definieren müssen. Eine Funktion wird durch das Schlüsselwort **FUNCTION** eingeleitet und mit dem Schlüsselwort **END** beendet. Die folgende Funktion **ABSVAL()** berechnet z.B. den Betrag einer Zahl:

```

REAL ABSVAL(r1)
REAL r1
IF (r1 > 0) THEN
  ABSVAL=r1
  RETURN
ELSE
  ABSVAL= (-1)*r1
  RETURN
ENDIF
END
    
```

Innerhalb eines Programmes können Sie **ABSVAL()** folgendermaßen aufrufen:

```

...
EXTERNAL ABSVAL
REAL ABSVAL
REAL r1 /-30.0/
...
PRINT *, 'Der Betrag von ', r1, ' ist ', ABSVAL(r1), '.'
...
    
```

Die Syntax für die **Typvereinbarung und Definition einer Funktion** **func1** vom Datentyp **type** lautet:

```

type FUNCTION func1(farg1, farg2, ...)
...
func1=expr
RETURN
END
    
```

Die Syntax für den **Aufruf einer Funktion** **func1** lautet:

```

...
[EXTERNAL func1]
type func1
...
... func1(aarg1, aarg2, ...) ...

```

Beachten Sie folgende Regeln:

Die **Formal-Parameter** farg1, farg2, ... werden beim Aufruf der Funktion durch die korrespondierenden **Aktual-Parameter** aarg1, aarg2, ... ersetzt.

Der **Rücksprung** in die aufrufende Programm-Einheit erfolgt entweder direkt nach einer RETURN Anweisung oder nach der abschließenden END Anweisung. Sie können mehrere RETURN Anweisungen verwenden.

Sie können Funktionen genauso wie Konstanten oder Variablen in Ausdrücken verwenden.

Eine Funktion sollte in der Regel **keine Nebeneffekte** verursachen, sondern ausschließlich einen Funktionswert und einen informativen Statuswert zurückliefern.

Sie sollten alle verwendeten Funktionen in einer **EXTERNAL** Anweisung als externe Prozeduren vereinbaren, u.a. um einen Überblick über die aufgerufenen Funktionen zu haben.

9.4. Vordefinierte Funktionen

In der FORTRAN Laufzeitumgebung, unter deren Kontrolle Ihre Programme ablaufen, sind standardmäßig u.a. die folgenden wichtigen mathematischen Funktionen vordefiniert:

Name	Bedeutung
ABS(x)	berechnet Betrag von x, d.h. x für REAL und x für COMPLEX
ACOS(x)	berechnet arccos(x)
ALOG(x)	berechnet $\ln(x) = \log_e(x)$
ASIN(x)	berechnet arcsin(x)
ATAN(x)	berechnet arctan(x)
COS(x)	berechnet cos(x)
EXP(x)	berechnet $\exp(x) = e^x$
INT(x)	konvertiert in eine ganze Zahl (Runden durch Abschneiden)
LEN(ch)	berechnet die Länge einer Zeichenkette
MAX(x,y,z,...)	berechnet Maximum von x, y, z, ...
MIN(x,y,z,...)	berechnet Minimum von x, y, z, ...
REAL(x)	konvertiert in eine reelle Zahl
SIGN(x)	berechnet Vorzeichen (Signum) von x
SIN(x)	berechnet sin(x)
SQRT(x)	berechnet Wurzel (Square Root) aus x
TAN(x)	berechnet tan(x)

Vordefinierte Funktionen werden im Standard als **intrinsische Funktionen** bezeichnet.

Die Syntax für den **Aufruf** einer intrinsischen Funktion func1 lautet:

```

[INTRINSIC func1]
...
... func1(aarg1, aarg2, ...) ...

```

...

9.5. Beispiel 18 - Satz des Pythagoras

Das folgende Programm SAMP9 berechnet aus den Längen von 2 Dreieckseiten die Länge der dritten:

```
PROGRAM SAMP9
  INTRINSIC SQRT
  DOUBLE PRECISION dS1, dS2, dS3
  PRINT *, 'Geben Sie die beiden Seitenlaengen ein:'
  READ *, dS1, dS2
  dS3=SQRT(dS1**2+dS2**2)
  PRINT *, 'Die dritte Seite hat die Laenge: ', dS3
  END
```

9.6. Beispiel 19 - Schaltjahr

Das folgende Programm SAMP10 überprüft, ob es sich bei einem Jahr um ein Schaltjahr handelt:

```
LOGICAL FUNCTION BLEAP(iYear)
  INTEGER iYear
  LOGICAL bY4, bY100, bY400
  bY4 =MOD(iYear,4).EQ. 0
  bY100=MOD(iYear,100).EQ. 0
  bY400=MOD(iYear,400).EQ. 0
  BLEAP=(bY4 .AND. .NOT. bY100) .OR. bY400
  RETURN
  END

PROGRAM SAMP10
  EXTERNAL BLEAP
  LOGICAL BLEAP
  INTEGER iYear
  PRINT *, 'Geben Sie das Jahr ein:'
  READ *, iYear
  IF ( BLEAP(iYear) ) THEN
    PRINT *, 'Das Jahr ', iYear, ' ist ein Schaltjahr.'
  ELSE
    PRINT *, 'Das Jahr ', iYear, ' ist kein Schaltjahr.'
  ENDIF
  END
```

(-> Übung 6)

9.7. Unterprogramme

Ein **Unterprogramm** realisiert einen Teilschritt eines Algorithmus.

Sie müssen ein Unterprogramm in einer **eigenen** Programm-Einheit definieren. Ein Unterprogramm wird durch das Schlüsselwort **SUBROUTINE** eingeleitet und mit dem Schlüsselwort **END** beendet. Sie können z.B. folgendes Unterprogramm MSG() zum Ausgeben einer Meldung definieren:

```

SUBROUTINE MSG(ch1)
CHARACTER*(*) ch1
CHARACTER*80 ch2 '*****'
PRINT *, ch2
PRINT *, '*** ', ch1
PRINT *, ch2
RETURN
END
    
```

Innerhalb eines Programmes können Sie das Unterprogramm MSG() folgendermaßen aufrufen:

```

CALL MSG('Ausfuehrung beginnt ...')
...
CALL MSG('Ausfuehrung beendet.')
    
```

Die Syntax für die **Definition eines Unterprogramms** sub1 lautet:

```

SUBROUTINE sub1(farg1, farg2, ...)
...
RETURN
END
    
```

Die Syntax für den **Aufruf eines Unterprogramms** sub1 lautet:

```

[EXTERNAL sub1]
...
CALL sub1(aarg1, aarg2, ...)
    
```

Beachten Sie folgende Regeln:

Die **Formal-Parameter** farg1, farg2, ... aus der Definition werden beim Aufruf des Unterprogramms durch die korrespondierenden **Aktual-Parameter** aarg1, aarg2, ... ersetzt.

Der **Rücksprung** in die aufrufende Programm-Einheit erfolgt entweder direkt nach einer RETURN Anweisung oder nach der abschließenden END Anweisung. Sie können mehrere RETURN Anweisungen verwenden.

Ein Unterprogramm liefert im Gegensatz zu einer Funktion nicht explizit einen Wert zurück. Sie können allerdings Variablen als Aktual-Parameter an ein Unterprogramm übergeben, im Unterprogramm verändern und danach die veränderten Werte in der aufrufenden Programm-Einheit weiterverwenden.

9.8.Beispiel 20 - Kreisberechnung

Das folgende Programm SAMP11 demonstriert, wie ein Unterprogramm über eine Statusvariable iFail zurückmelden kann, ob seine Ausführung erfolgreich war. Das Haupt-Programm wertet die Variable iFail aus und kann eine entsprechende Meldung ausgeben:

```

SUBROUTINE CIRCLE(rCircleRadius, iFail)
REAL rCircleRadius
INTEGER iFail
REAL rPI, rCircleArea, rCircleCirc
PARAMETER (rPI=3.141592653)
IF (rCircleRadius < 0) THEN
  PRINT *, 'CIRCLE: Negatives Argument.'
  iFail = -1
  RETURN
ELSE
  rCircleArea = rPI * (rCircleRadius**2)
  rCircleCirc = 2 * rPI * rCircleRadius
  PRINT *, 'Die Kreisflaeche betraegt: ', rCircleArea
  PRINT *, 'Der Kreisumfang betraegt: ', rCircleCirc
  iFail = 0
RETURN
ENDIF
END

PROGRAM SAMP11
EXTERNAL CIRCLE
REAL rCircleRadius
INTEGER iFail
PRINT *, 'Geben Sie den Kreisradius ein:'
READ *, rCircleRadius
CALL CIRCLE(rCircleRadius, iFail)
IF ( iFail .NE. 0 ) THEN
  STOP 'MAIN: Fehler in CIRCLE.'
ENDIF
END

```

9.9. Prozeduren mit variablen Feldern und Zeichenketten

Sie können an eine Prozedur ein **Feld** oder eine **Zeichenkette mit variabler (vererbter) Länge** übergeben. Es gibt hierfür zwei Möglichkeiten:

* als letzte Dimension:

Sie definieren die **letzte** Dimension eines Feldes oder einer Zeichenkette in einer Prozedur mit einem *. In diesem Fall wird beim Aufruf die aktuelle Länge **automatisch** übergeben:

Definition:

```

REAL FUNCl(rA, rB, chC)
REAL rA(*), rB(*)
CHARACTER*(*) chC

```

Aufruf:

```

REAL rArray(10), rArray2(10)
CHARACTER*80 chl
...
FUNCl(rArray1, rArray2, chl)

```

Explizite Übergabe der Dimension in weiteren Parametern

Sie definieren die Dimensionen eines Feldes über weitere Formal-Parameter bzw. über Variablen in einem COMMON Block (siehe unten). In diesem Fall müssen Sie beim Aufruf die aktuellen Dimensionen übergeben:

Definition:

```
REAL FUNC1(rA, rB, iM, iN, iK, chC, iL)
REAL rA(iM), rB(iN)
CHARACTER*(iL) chC
```

Aufruf:

```
FUNC1(rArray1, rArray2, 10, 10, ch1, LEN(ch1))
```

9.10. Beispiel - Vektormultiplikation

Das folgende Programm SAMP12 nutzt eine Funktion VECMUL(), die Felder variabler Länge verarbeitet. Die Feldlänge wird im Formal-Parameter iN übergeben:

```
REAL FUNCTION VECMUL(rA, rB, iN)
REAL rA(iN), rB(iN)
INTEGER iN, i1
VECMUL = 0.0
DO i1=1, iN
    VECMUL = VECMUL + rA(i1)*rB(i1)
END DO
RETURN
END

PROGRAM SAMP12
EXTERNAL VECMUL
INTEGER iN
PARAMETER (iN=5)
REAL rA(iN) /1,1,1,1,1/, rB(iN) /1,1,1,1,1/
PRINT *, 'Die Vektormultiplikation ergibt: ', VECMUL(rA, rB, iN)
END
```

9.11. Prozeduren mit statischen Variablen

Die in einer Prozedur definierten Variablen (lokale Variablen) werden nach dem Rücksprung in die aufrufende Einheit nicht gesichert und sind deshalb beim nächsten Aufruf nicht mehr definiert.

Eine **SAVE Anweisung** für eine lokale Variable stellt sicher, daß der letzte gültige Wert dieser lokalen Variablen für den nächsten Aufruf erhalten bleibt (statische Variable). Die Variable iCount können Sie z.B. folgendermaßen in einer Prozedur SIMPLE als Aufrufzähler nutzen:

```
SUBROUTINE SIMPLE()
INTEGER iCount /0/
SAVE iCount
iCount = iCount +1
PRINT *, 'SIMPLE: Aufruf Nr. ', iCount
END
```

Die Syntax für eine **SAVE Anweisung** in einer Prozedur proc1 lautet folgendermaßen.

```
procedure proc1...
...
SAVE [var1, var2, ...]
...
```

Beachten Sie,

daß die SAVE Anweisung allein **alle** lokalen Variablen und COMMON Blöcke (siehe unten) als statisch definiert.

9.12. COMMON Blöcke

Ein **COMMON Block** dient zum Austausch von Daten zwischen unterschiedlichen Programm-Einheiten.

Sie können eine Variable aus einem COMMON Block in jeder Programm-Einheit auswerten oder verändern, in der der COMMON Block vereinbart ist; d.h. die in einem COMMON Block enthaltenen Variablen können **global** von allen Programm-Einheiten zugänglich (globale Variablen).

Sie können z.B. mit folgender Vereinbarung in allen Programm-Einheiten einen COMMON Block mit dem symbolischen Namen BLOCK1 vereinbaren, der eine 100x100 Matrix enthält:

```
INTEGER iN
PARAMETER (iN=100)
REAL rMatrix(iN,iN)
COMMON /BLOCK1/ rMatrix(iN,iN)
```

Die Syntax für die **Vereinbarung eines COMMON Blocks** block1 in einer Programm-Einheit lautet:

```
type var1 [, var2 ...]
COMMON /block1/ var1 [,var2 ...]
[SAVE /block1/]
```

Da ein COMMON Block von mehreren Programm-Einheiten bearbeitet werden kann, können Sie die Anfangswertzuweisung nicht wie gewohnt vornehmen, sondern benötigen hierzu eine eigene **BLOCK DATA** Programm-Einheit.

Die Syntax für die **Anfangswertzuweisung für Variablen in einem COMMON Block** block1 lautet:

```
BLOCK DATA b1
type var1 [, var2 ...]
COMMON /block1/ var1 [,var2 ...]
END
```

Beachten Sie folgende Regeln:

Sie müssen einen COMMON Block in allen Programm-Einheiten mit den selben **Datentypen** und mit der selben **Reihenfolge** der Variablen definieren, wobei allerdings die Wahl der **Namen** frei ist.

Eine **Anfangswertzuweisung** für Variablen in einem COMMON Block darf nur in einer BLOCK DATA Programm-Einheit erfolgen.

9.13. Beispiel 21 - Globale Variablen in einem COMMON Block

Das folgende Programm SAMP13 demonstriert die Verwendung eines COMMON Blocks in 3 Programm-Einheiten. Die Variable r1 steht allen 3 Programm-Einheiten zur Verfügung:


```

SUBROUTINE SUB1()
REAL r1
COMMON /BLOCK1/ r1
PRINT *, 'SUB1: r1 = ', r1
r1 = 1
RETURN
END

SUBROUTINE SUB2()
REAL r1
COMMON /BLOCK1/ r1
PRINT *, 'SUB2: r1 = ', r1
r1 = 2
RETURN
END

PROGRAM SAMP13
REAL r1
COMMON /BLOCK1/ r1
r1=0
PRINT *, 'MAIN: r1 = ', r1
call SUB1
PRINT *, 'MAIN: r1 = ', r1
call SUB2
PRINT *, 'MAIN: r1 = ', r1
END
    
```

(-> Übung 7)

9.14. Zusammenfassung

Sie haben nun folgende weiteren Sprachelemente kennengelernt:

- **BLOCK DATA Programm-Einheiten**
- **COMMON Blöcke**
- **EXTERNAL Anweisungen**
- **Formelfunktionen**
- **Funktionen**
- **INTRINSIC Anweisungen**
- **Prozeduren mit Feldern und Zeichenketten variabler Länge**
- **RETURN Anweisungen**
- **SAVE Anweisungen**
- **Unterprogramme**

Auf Grundlage dieser Sprachelemente lassen sich Programme folgender Bauart entwickeln:

Beginn des Haupt-Programmes	PROGRAM pgm
Vereinbarung von externen und intrinsischen Prozeduren	EXTERNAL ... INTRINSIC ...
Typvereinbarung von Variablen, Feldern und symbolischen Konstanten	INTEGER iVar1, iVar2, iArray(30)
Definition von symbolischen Konstanten	PARAMETER (rCONST1=2.843)
Definition von COMMON Blöcken	COMMON iArray(30)
Definition von Formelfunktionen	AVG(a,b)=(a+b)/2
Anfangswertzuweisungen	DATA ...
Einlesen von Variablen und Feldern	READ *, rVar1
Zuweisungen	rVar2 = rCONST1 * rVar1
DO Anweisungen (Schleifen)	DO ...
IF Anweisungen (Verzweigungen)	IF ...
Aufruf von Prozeduren	CALL ...

Ausgeben von Ausdrücken	PRINT *, 'rVar2 = ', rVar2
GOTO Anweisungen (Sprunganweisungen)	GOTO ...
Ende des Hauptprogramms	END

Beginn des Unterprogramms	SUBROUTINE sub1(...)
Vereinbarungen	...
Definition statischer Variablen	SAVE ...
ausführbare Anweisungen	...
Rücksprung zur aufrufenden Einheit	RETURN
Ende des Unterprogramms	END

Beginn der Funktion	REAL FUNCTION func1(...)
Vereinbarungen	...
Definition statischer Variablen	SAVE ...
ausführbare Anweisungen	...
Rücksprung zur aufrufenden Einheit	RETURN
Ende der Funktion	END

Beginn des Datenblockes	BLOCK DATA bdata1
Typvereinbarungen mit Anfangswertzuweisungen	REAL ... /.../
Vereinbarungen von COMMON Blöcken	COMMON /.../ ...
Ende des Datenblockes	END

Beachten Sie, daß Sie in FORTRAN Programm immer folgende Reihenfolge einhalten müssen:

Kommentar	PROGRAM / FUNCTION / SUBROUTINE / BLOCK DATA		
		PARAMETER	IMPLICIT andere Vereinbarungen (z.B. REAL ...)
		DATA	Formelfunktionen ausführbare Anweisungen (z.B. READ *, ...)
	END		

Beachten Sie ferner die Unterschiede zwischen **globalen** und **lokalen** Variablen und bei letzteren zwischen **statischen** und **automatischen** Variablen:

Variable		
global (COMMON ...)	lokal (bezüglich einer Programm-Einheit)	
	statisch (SAVE)	automatisch

10.Formatierte Ein- und Ausgabe und Dateiverarbeitung

In diesem Kapitel lernen Sie die formatierte Ein- und Ausgabe kennen. Ferner lernen Sie die Möglichkeit kennen, Daten auf externen Medien (z.B. Dateien) zu speichern bzw. Daten von externen Medien einzulesen.

10.1. Formatierte Ein- und Ausgabe

Bei der bisher behandelten **listengesteuerten Ausgabe** mit PRINT * benutzt der FORTRAN Sprachübersetzer (Compiler) vom Datentyp abhängige Standarddarstellungen für die auszugehenden Ausdrücke in der Ausgabeliste. Im Gegensatz hierzu können Sie bei der **formatierten Ausgabe** die Darstellung **zeichengenau** vorschreiben..

Ebenso führt der FORTRAN Sprachübersetzer (Compiler) bei der **listengesteuerten Eingabe** mit READ * automatisch eine Umwandlung der Eingabedaten in den Datentyp der korrespondierenden Variablen in der Eingabeliste durch. Im Gegensatz hierzu müssen Sie bei der **formatierten Eingabe** die Darstellung **zeichengenau** angeben.

Eine **formatierte Ausgabe** erfolgt anhand einer zeichengenauen Formatbeschreibung der auszugehenden Ausdrücke. Sie können z.B. mit folgender WRITE Anweisung, die auf eine Formatbeschreibung an der Sprungmarke 100 verweist, die Variablen i1, i2 und i3 mit jeweils 10 Zeichen auf der Standardausgabe (* für Bildschirm) ausgeben:

```

                INTEGER i1 /5/, i2 /-20/, i3 /-5000/
                WRITE(*,100) i1, i2, i3
100            FORMAT(I10, I10, I10)
    
```

Die Ausgabe umfaßt genau 30 Zeichen, da die Formatbeschreibung aus 3 Datenfeldbeschreibungen für INTEGER Ausdrücke mit Länge 10 besteht. (Zur Verdeutlichung sind im folgenden Leerzeichen durch ein kleines b symbolisiert.)

```

bbbbbbbbbb5bbbbbbb-20bbbb-5000
|          |          |
123456789012345678901234567890
I10        I10        I10
    
```

Sie können folgende Datenfeldbeschreibungen für die Datentypen CHARACTER, INTEGER, REAL, DOUBLE PRECISION (DP) und LOGICAL verwenden:

Deskriptor	Datentyp	Beispiel	Ausgabe
A	CHARACTER	A-> 'Hallo'	Hallo
Aw	CHARACTER	A10-> 'Hallo'	_____Hallo
Iw	INTEGER	I5-> 55	_____55
Iw.m	INTEGER	I5.3 -> 55	____055
Fw.d	REAL, DP	F7.3-> 23.4	__23.400
Ew.d[Ee]	REAL, DP	E8.2-> 23.4	__0.23E02
Lw	LOGICAL	L1 -> .TRUE.	T

Erläuterungen:

w gibt jeweils die **Datenfeldweite** an, d.h. die gesamte Länge des Ausgabefeldes. d gibt die Anzahl der **Dezimalstellen** an, e die Anzahl der **Ziffern im Exponenten** und m die Anzahl der ausgegebenen Ziffern (ggfs. werden führende Nullen vorangestellt).

Für einen **COMPLEX** Ausdruck benötigen Sie **zwei** REAL Datenfeldbeschreibungen, jeweils eine für Real- und Imaginärteil.

Die Datenfeldbeschreibungen sind wiederholbar, indem Sie ihnen einen **Wiederholungsfaktor** n vorstellen. Z.B. ist 3I10 äquivalent zu I10, I10, I10.

Weitere Deskriptoren in einer Formatanweisung dienen Ihnen zur Positionierung und zur Ausgabe von Zeichenketten-Konstanten. Sie können diese Deskriptoren an beliebiger Stelle zwischen den Datenfeldbeschreibungen einfügen:

Deskriptor	Bedeutung
nX	setzt um n Stellen vor
n('abc')	gibt die Zeichenkette 'abc' n-mal hintereinander aus
Tn	positioniert absolut auf Spalte n
TLn	setzt um n Stellen vor
TRn	setzt um n Stellen zurück

Die Syntax für eine **WRITE Anweisung mit Formatsteuerung** lautet:

```

WRITE([unit=]n, [fmt=]label [...]) expr1 [,expr2 ...]
label  FORMAT(d1 [,d2 ...])

n=0, ..., 99, *
d1, d2= Iw, Ew.dEe, Lw, Fw.d, Aw, ...
    
```

Die formatierte Eingabe mit der READ Anweisung mit Formatsteuerung erfolgt analog zur formatierten Ausgabe.

Die Syntax für eine **READ Anweisung mit Formatsteuerung** lautet:

```

READ([unit=]n, [fmt=]label [...]) var1 [,var2 ...]
label  FORMAT(df1 [,df2 ...])

n=0, ..., 99, *
    
```

Beachten Sie folgende Regeln:

Die Anzahl der Datenfeldbeschreibungen und der Variablen sollten übereinstimmen. Bei zu wenigen Eingabedaten in einer Eingabezeile (Datensatz) werden die restlichen Eingabedaten aus der nächsten Eingabezeile gelesen. Bei zu vielen Eingabedaten werden die restlichen ignoriert.

Falls eine Datenfeldbeschreibung das Einlesen einer Variablen nicht zulässt, wird - systemabhängig - das Einlesen abgebrochen oder es wird ein abgeschnittener Wert zugewiesen. Beim Ausgeben eines Ausdrucks wird ggfs. gerundet. Zu große Ausdrücke werden nicht ausgegeben, sondern stattdessen wird das Ausgabefeld mit Sternen * aufgefüllt.

Bei Erreichen des letzten Datensatzes (Dateiende) können Sie über den **END** Parameter in der READ Anweisung (END=label) zu einer Sprungmarke label verzweigen, an der die Ausführung fortgesetzt wird. Z.B. verzweigt ein Programm aufgrund der Anweisung READ(..., END=1000) zur Marke 1000, wenn das Dateiende erreicht ist.

Im Fehlerfall können Sie die weiteren Parameter **ERR** und **IOSTAT** verwenden, um eine Fehlerbehandlung durchzuführen (siehe [1]).

10.2. Beispiel 22 - Ausgeben einer Tabelle

Das folgende Programm SAMP14 gibt eine formatierte Tabelle aus:

```

PROGRAM SAMP14
INTEGER i1
REAL r1,rPi /3.141/
WRITE(*,1000)
WRITE(*,1010)
WRITE(*,1000)
DO i1=1, 10
  WRITE(*,1020) (i1/100.0), rPi*(i1/100.0)**2
ENDDO
WRITE(*,1000)
1000 FORMAT(1X,'+', 12('-'),
1010 FORMAT(1X,'|', 'Radius', '|', 'Flaeche', '|')
1020 FORMAT(1X,'|', 'E10.4', '|', 'E10.4', '|')

END

```

10.3. Dateiverarbeitung

Ein FORTRAN Programm liest und schreibt Daten von **Einheiten**, die über eine Einheitennummer bezeichnet werden. Die Art der Verknüpfung von Einheitennummern mit Dateien oder anderen Medien wie z.B. Magnetbandstationen oder Druckern ist systemabhängig (siehe unten).

Sie können den Zugriff auf Einheiten in einem FORTRAN Programm mit folgenden Anweisungen realisieren, wobei Sie die Reihenfolge OPEN - READ/WRITE - CLOSE einhalten müssen:

INQUIRE (FILE='...', ...)	fragt den Status einer Datei oder einer Einheit ab
OPEN (9, FILE='INDATA')	verbindet eine Datei mit der Einheit 9
READ (9, ...)	liest einen Datensatz von Einheit 9
OPEN (10, FILE='OUTDATA')	verbindet eine Datei mit der Einheit 10
WRITE (10, ...)	schreibt einen Datensatz auf Einheit 10
CLOSE (9, ...)	schließt die Verbindung zu Einheit 9
CLOSE (10, ...)	schließt die Verbindung zu Einheit 10

Beachten Sie folgende Regeln:

Standardmäßig sind die Einheitennummer 5 mit der Standardeingabe (Tastatur) und 6 mit der Standardausgabe (Bildschirm) verknüpft. In READ und WRITE Anweisungen kann deshalb anstelle der Einheitennummer auch ein * für Einheit 5 bzw. Einheit 6 angegeben werden.

Verknüpfung von Einheiten und Dateien unter VM/CMS:

Die Verknüpfung von Einheitennummern und Dateien erfolgt (spezifisch für VM/CMS) zweistufig, indem zunächst eine Einheitennummer mit einem logischen Namen und dieser auf Betriebssystemebene mit einem Dateinamen verknüpft wird. Im folgenden Beispiel wird die Einheitennummer 8 mit der Datei INPUT DATA A verknüpft:

Betriebssystem-Kommando:

```
FILEDEF INDATA DISK INPUT DATA A
```

FORTRAN Quellprogramm:

```
OPEN(unit=9, FILE='INDATA')
```

Es ergibt sich in diesem Fall folgende Verknüpfung:

Einheitennummer ->	logischer Name ->	Dateiname
9	INDATA	INPUT DATA A

Die Syntax für eine **OPEN Anweisung** lautet:

```
OPEN([UNIT=]n, [FILE=]'...', BLANK='NULL', IOSTAT=i1 [...])
```

```
n=0, ..., 99, *
```

Die Syntax für eine **CLOSE Anweisung** lautet:

```
CLOSE([UNIT=]n, IOSTAT=i1 [...])
```

```
n=0, ..., 99, *
```

10.4. Beispiel 23 - Einlesen aus einer Datei

Das folgende Programm SAMP15 schreibt zunächst 10 Datensätze in eine Datei DAT1, die über ein **FILEDEF** Kommando (nur VM/CMS!) definiert sein muß. Die Anweisung **REWIND** positioniert den Dateizeiger auf den Dateianfang. Danach werden die 10 Datensätze gelesen:

```
PROGRAM SAMP15
INTEGER i1, iMAX
PARAMETER(iMAX=10)
REAL r1
OPEN(UNIT=9, FILE='DAT1', BLANK='NULL', STATUS='NEW')
REWIND(9)
DO i1=1, iMAX
WRITE(9,1000) i1, i1*100.0
END DO
REWIND(9)
DO i1=1, iMAX
READ(9,1000) i1, r1
PRINT *, 'i1 = ', i1, ' i1*100.0 = ', r1
END DO
1000 FORMAT(I5, E10.2)
CLOSE(9)

! Vergessen Sie nicht, DAT1 unter VM/CMS
! mit dem Kommando FILEDEF DATA DISK fn ft fm mit
! einer Datei zu verknuepfen, z.B. mit TEMP1 DAT1 A.
```

(-> Übung8)

10.5. Zusammenfassung

Sie haben nun folgende weiteren Sprachelemente zur Ein- und Ausgabe kennengelernt:

- **CLOSE Anweisungen**
- **Dateifeldbeschreibungen**
- **Formatbeschreibungen**
- **INQUIRE Anweisungen**
- **OPEN Anweisungen**
- **READ Anweisungen mit Formatbeschreibung**
- **REWIND Anweisungen**
- **REWIND Anweisungen**
- **WRITE Anweisungen mit Formatbeschreibung**

Auf Grundlage dieser Sprachelemente lassen sich Dateizugriffe zum Einlesen und Ausgeben folgender Bauart entwickeln:

	...
Abfragen des Dateistatus	INQUIRE(FILE='INDATA', ...)
Öffnen einer Datei	OPEN(unit=8, FILE='INDATA', iostat=i1)
Formatiertes Einlesen aus einer Datei	READ(unit=8, fmt=1000) i1, i2
Öffnen einer Datei	OPEN(unit=9, FILE='OUTDATA', iostat=i1)
Zurückspulen einer Datei	REWIND(9)
Formatiertes Ausgeben in eine Datei	WRITE(unit=9, fmt=2000) r1, r2
Schließen einer Datei	CLOSE(unit=9, iostat=i1)
	...
Formatanweisungen	1000 FORMAT(I10,I10)
	...

11. Bemerkungen

In diesem letzten Kapitel erhalten Sie kurze Hinweise zur Programmiermethodik in FORTRAN, zu nicht behandelte Sprachelemente und zu wünschenswerten Erweiterungen der Sprache FORTRAN.

11.1. Programmiermethodik

Sie können in der Regel die Qualität Ihrer Programme verbessern, wenn Sie bei der Programm-entwicklung folgende Regeln einhalten:

- Kommentieren Sie Ihr Programm ausreichend.
- Verwenden Sie symbolische Konstanten.
- Vereinbaren Sie alle Variablen und geben Sie Ihnen einen sinnvollen Namen und Anfangswert.
- Strukturieren Sie Ihr Programm durch Unterprogramme und Funktionen.
- Heben Sie Blöcke im Programm durch Einrückungen hervor.

Falls sich Ihr Programm "sonderbar" verhält, grenzen Sie die Problemursache (z.B. durch "Verkleinerung" des Programms) ein und analysieren Sie die übriggebliebenen Anweisungen durch explizite Ausgabe von aktuellen Werten (PRINT *) oder mit Hilfe eines FORTRAN Debuggers.

11.2. Weitere Sprachelemente

Folgende Sprachelemente sind **nicht** behandelt worden:

- arithmetische IF Anweisungen
- ASSIGNED GOTO Anweisungen
- BACKSPACE/ENDFILE Anweisungen
- COMPUTED GOTO Anweisungen
- DIMENSION Anweisungen
- ENTRY Anweisungen
- EQUIVALENCE Anweisungen
- generische und spezifische Funktionen
- IMPLICIT Anweisungen
- interne Dateien
- logische IF Anweisungen
- Prozeduren als Parameter
- RETURN * Anweisungen
- Teilfelder und Teilzeichenketten
- Zeichenketten-Verknüpfung //

11.3. Wünschenswerte Erweiterungen

Sie sollten in FORTRAN im Vergleich zu anderen Programmiersprachen folgende Sprachelemente und Möglichkeiten vermissen:

- DO WHILE Anweisung
- dynamische Speicherverwaltung
- INCLUDE Direktive
- problemorientierte Datentypen (Strukturen)
- SELECT Anweisung
- Standard-Bibliothek für Betriebssystemaufrufe
- strenge Typprüfung

Die Weiterentwicklung von FORTRAN 77, die als Fortran 90 bezeichnet wird, bietet u.a. die aufgelisteten Sprachelemente und Möglichkeiten an. Auch einige verfügbare FORTRAN 77 Compiler bieten über den Standard hinausgehende (und damit leider auch nicht portierbare) Spracherweiterungen und Möglichkeiten an.

11.4. Weitere Werkzeuge

Für anspruchsvollere Projekte benötigen Sie zusätzliche Werkzeuge:

- Syntaxchecker
- Interaktiver Debugger
- Bibliotheksverwalter
- MAKE Projektverwalter

11.5. Weitere Bibliotheken

Für anspruchsvollere Projekte benötigen Sie neben der FORTRAN Laufzeitbibliothek, die u.a. die intrinsischen Funktionen enthält, in der Regel weitere spezialisierte Bibliotheken in folgenden Anwendungsgebieten:

- Grafik
- Numerik und Statistik
- Betriebssystemaufrufe

12. Übungen

12.1. Übung 1: Algorithmen, Flußdiagramme und Pseudo-Code

1. Überprüfen Sie, ob die folgenden Algorithmen die geforderten 4 Eigenschaften besitzen und verbessern Sie die Algorithmen, falls notwendig:

a) Algorithmus "Übersetzung"

1. Lesen Sie ein Wort ein, das übersetzt werden soll.
2. Suchen Sie eine Übersetzung für das Wort.

b) Algorithmus "Ampel"

1. Wenn die Ampel "rot" zeigt, warten Sie solange, bis sie "grün" zeigt.
2. Fahren Sie über die Kreuzung.

c) Algorithmus "Lottozahlen"

1. Kreuzen Sie 6 Zahlen pro Feld an.
2. Reichen Sie Ihren Lottoschein ein.
3. Falls Sie nicht gewonnen haben, gehen Sie zu 1.

2. Entwerfen Sie Algorithmen für folgende Aufgabenstellungen:

- a) Suchen einer Telefonnummer
- b) Suchen der größten Zahl in einer Liste von Zahlen
- c) Lösen der Gleichung $ax+b=0$

3. Welche Kriterien bestimmten den von Ihnen gewählten Detaillierungsgrad für Aufgabe 2? Können Sie den Detaillierungsgrad vergrößern/verkleinern?

4. Markieren Sie in Ihren Algorithmen zu Aufgabe 2 jeweils die Stellen, die für eine Beendigung nach **endlich** vielen Schritten sorgen.

5. Testen Sie den Pseudo-Code "Summation der Zahlen von 1 bis N" mit folgenden Eingabedaten: N=1, N=3, N=-2, N=2,5

6. Vereinfachen Sie den Pseudo-Code aus Aufgabe 6. (Hinweis: $1+2+\dots+N=N*(N+1)/2$)

7. Erstellen Sie für den Algorithmus "Lösen der Gleichung $ax+b=0$ " ein Flußdiagramm.

Hinweis: Benutzen Sie die grafischen Symbole für START, END, READ, PRINT, STATEMENT, CONDITION und kennzeichnen Sie die Richtung, in der das Flußdiagramm durchlaufen werden soll, mit Pfeilen.

8. Erstellen Sie für den Algorithmus "Suchen des größten Elementes in einer Liste von Zahlen" ein Flußdiagramm.

12.2. Übung 2: Entwicklungsumgebung

Im folgenden wird die Entwicklungsumgebung auf dem System RZUNIOS IBM 3090 unter VM/CMS als Grundlage genommen (siehe z.B. [2]). Die Entwicklungsumgebungen auf anderen Systemen sind jedoch relativ ähnlich, so daß Sie sich in der Regel problemlos umstellen können.

1. Welche Werkzeuge benötigen Sie in welcher Reihenfolge, um ein ausführbares Programm zu erzeugen?

2. Erstellen Sie mit einem Editor (hier: **XEDIT**) folgendes FORTRAN Quellprogramm:

```
*2345678901234567890123456789012345678901234567890
PROGRAM HELLO
PRINT *, 'Hello World'
END
```

3. Übersetzen Sie das Programm mit einem FORTRAN Sprachübersetzer (Compiler) (hier: **FORTVS2**) und überprüfen Sie das Listing.

4. Laden Sie das Programm und führen Sie es aus (hier: Lader **LOAD** und Programm-Starter **START**).

5. Fügen Sie einen Kommentarkopf in das Programm ein.

6. Bauen Sie absichtlich einen Fehler in das Programm ein, und übersetzen Sie es erneut. (Hinweis: Ersetzen Sie z.B. PRINT durch PRITT oder beginnen Sie statt in Spalte 7 in Spalte 6.)

7. Drucken Sie Ihr Quellprogramm am Systemdrucker aus.

12.3. Übung 3: Konstanten, Variablen, Ausdrücke und Zuweisungen

1. Entscheiden Sie, ob die folgenden arithmetischen Konstanten für den genannten Datentyp gültig sind und tragen Sie den entsprechenden numerischen Wert im Kasten rechts daneben ein. Falls das Literal keine arithmetische Konstante repräsentiert, geben Sie im Kasten eine kurze Begründung:

a) INTEGER:

-3		-300.	
300 000 0		10000000000000	
0		-0	
0xFF		Z'FF'	
20 000		300000L	
Hundert		'200'	

b) REAL:

-3.9		- 3.9	
4.5E20		4.5E200	
.1		1.E23	
1.0E-0.5		4E10	
0.0		4E-5.	

c) COMPLEX:

-3.9		(-3.9,1)	
(4.5E20)		4.5E2+i1.0	
(.1,.1)		(1.E23,-1E23)	
(1.0E-0.5,1)		(4.E10, -4)	
0		.	

2. Ordnen Sie die folgenden Literale einem arithmetischen Datentyp zu. Falls es sich um keine gültige arithmetische Konstante handelt, geben Sie eine kurze Begründung:

23,5		1E14	
1		7,0	
3x10 ⁷		121000000000	
8E17		0.3E-7.0	

3. Für eine Konstante vom Datentyp INTEGER sollen 2 Bytes Speicherplatz zur Verfügung stehen, Bit 0 wird dabei zur Darstellung des Vorzeichens s verwendet, die restlichen 15 Bits zur Darstellung des Betrags im Dualsystem: |snnnnnnn| |nnnnnnnn|

- a) Stellen Sie die Zahlen 5 und (-10) in obiger Form dar.
- b) Welcher Wertebereich kann dargestellt werden?

4. Welche symbolischen Namen sind zulässig, welche nicht? Welche Namen sind gleich, d.h. bezeichnen das selbe Objekt?

Hinweis:

Es gibt von Compiler zu Compiler Unterschiede hinsichtlich der zulässigen Gesamtlänge von Namen und der Möglichkeit der Kleinschreibung. Gehen Sie in dieser Übung zunächst von 7 Zeichen und Großschreibung (i), dann von erlaubter Kleinschreibung (ii) und dann von unterschiedlichen Namen bei Groß- und Kleinschreibung und maximal 32 Zeichen langen Namen aus (iii.)

- | | (i) | (ii) | (iii) |
|----|-------------------|------|-------|
| a. | EINSEHRLANGERNAME | | |
| b. | INTEGER | | |
| c. | Name | | |
| d. | NAME | | |
| e. | Mein Name | | |
| f. | STOP! | | |
| g. | A | | |
| h. | i100 | | |
| i. | 100i | | |
| j. | BBC-2 | | |
| k. | X4.9 | | |
| l. | 'AB' | | |

5. Werten Sie folgende Ausdrücke durch Setzen von Klammern per Hand aus und überprüfen Sie Ihre Auswertung im Programm SAMP16 (ggfs. später). Können Sie sich den Unterschied zwischen (ii) und (iii) erklären?

- (i) $5 + 3 * 8 / 4 * 2$ (ii) $3 / 10$ (iii) $3 / 10.0$ (iv) $3 * 8 / 2 ** 2$

```
PROGRAM SAMP16
PRINT *, '5+3*8/4*2 = ', 5+3*8/4*2
PRINT *, ...
END
```

6. Welche Werte ergeben folgende arithmetischen Ausdrücke? Sind sie ggfs. unzulässig?

- | | |
|------------|----------------|
| 6/3 | 8/3 |
| -8/3 | -1**3 |
| (-1)**3.5 | 2*-3 |
| (2+4)(4+5) | 3+(2*8)+4) |
| (-4)**(-3) | 2**3**2 |
| (2**3)**2 | (8*3)+5)*(7-4) |

7. Schreiben Sie ein Programm SAMP17, daß zwei Zahlen einliest und die Summe, die Differenz, den Mittelwert und das Produkt berechnet und ausgibt:

```
PROGRAM SAMP17
REAL rVar1 /0/, rVar2/0/
PRINT *, "Geben Sie zwei Zahlen ein!"
READ rVar1, rVar2
...
PRINT *, 'Die eingegebenen Zahlen waren: ', rVar1, rVar2
PRINT *, 'Summe: ', ...
...
END
```

Hinweis:

Benutzereingaben sind "unberechenbar" und sollten auf Gültigkeit überprüft werden. Sie werden das an geeigneter Stelle nachholen. Wie reagiert das Programm z.B. auf die Eingaben rVar1=0 und rVar2=0?

9. Schreiben Sie ein Programm zur Umwandlung von Fahrenheit (F) in Celsius (C). Die Variable rFahrenheit, die in Celsius umgerechnet werden soll, soll interaktiv eingelesen werden.

Hinweis:

$$F=(9.0/5.0)*C+32.0$$

12.4. Übung 4: DO Anweisungen und Felder

1. Wie oft werden folgende DO Anweisungsschleifen durchlaufen:

- a) DO i1=1, 101
- b) DO i1=2, 10, 2
- c) DO i1=1, 10, -1
- d) DO r1=0.0, 1.0, 0.1
- e) DO r1=0.1, 0.3, -0.1

2. Schreiben Sie ein Programm SAMP18, das die Fakultät $N!$ berechnet. Die positive ganze Zahl N soll interaktiv eingelesen werden.

Hinweis:

$$N! = 1 * 2 * 3 * \dots * N$$

3. Schreiben Sie ein Programm SAMP19, das den Mittelwert, die Summe der Quadrate und die Summe der Kehrwerte der Zahlen von 1 bis 100 berechnet.

4. Wieviele Elemente besitzen folgende Felder?

- (i) rVector(20), (ii) rGrid(-10:5,-20:1,1:11), (iii) rMatrix(5,5), (iv) bArray(1:3,4:7,7:4)

5. Es sollen folgende Vereinbarungen gelten:

```
REAL rVector(10) /10,9,8,7,6,5,4,3,2,1/
INTEGER i1 /5/
INTEGER i2 /2/
```

Welche Werte besitzen folgende Elemente bzw. welche Werte sind nicht definiert?

- a) rVector(3), b) rVector(i1-4.2), c) rVector(i1+i2)
- d) rVector(i1**2), e) rVector(rVector(rVector(i1)))

6.(*). Schreiben Sie ein rudimentäres Programm SAMP20, das ein beliebiges Polynom 2. Grades ($a*x^2+b*x+c$) durch Approximation an $(n+1)$ Stützstellen $h(i,n)=x(1)+i/(n*(x(2)-x(1)))$, $i=0,\dots,n$ integriert. Setzen Sie zunächst $n=10$.

Hinweis:

Sie benötigen u.a. folgende Variablen:

x(1)=rLeftB	linke Intervallgrenze
x(2)=rRightB	rechte Intervallgrenze
h(i,n)=iStep	Schrittweite
a=rA	Koeffizient vor x^2
b=rB	Koeffizient vor x
c=rC	Koeffizient vor 1

7. (*) Verbessern Sie das Programm SAMP20 (beliebige Schrittweiten, Ober- und Untersummen, Vergleich mit dem Wert des exakten Integrals, Trapeze statt Treppen, Abbruchkriterium, Polynome höheren Grades, ...).

8. (*) Schreiben Sie ein Programm SAMP21, daß das Matrixprodukt zweier 3x3-Matrizen berechnet.

9. (*) Mehrdimensionale Felder werden in FORTRAN Spaltenweise abgelegt. Es ist in FORTRAN erlaubt, in einem Unterprogramm ein mehrdimensionales Feld über nur einen Index anzusprechen, d.h.. rA(iN, iM) entspricht rB(iN*iM). Wie muß i3 gewählt werden, damit die Elemente rA(i1,i2) und rB(i3) gleich sind?

12.5. Übung 5: Logische Ausdrücke und Zuweisungen, IF Anweisungen

1. Sei $iN=5$ und $iM=3$. Welche der folgenden Vergleiche liefern den Wahrheitswert .TRUE.?

- a) $(iN > 0) .AND. (iM \neq 4)$
- b) $(iN .EQ. 0) .OR. (iM .GR. 2)$
- c) $(iN .LE. 5) .AND. (iM .GE. 3)$

2. Schreiben Sie ein Programm SAMP22, das für 2 logische Variablen l1 und l2 Wahrheitstafeln für die Operatoren .AND. und .OR. aufstellt.

3. Schreiben Sie ein Programm SAMP23, das 3 Zahlen einliest und dann die größte und die kleinste eingelesene Zahl berechnet.

4. (*) Schreiben Sie ein Programm SAMP24, das überprüft, ob eine eingelesene positive Zahl N eine Primzahl ist.

Hinweis:

Verwenden Sie die Funktion MOD(N,i), die den Rest bei Division berechnet, z.B. liefert. Z.B. liefert MOD(5,2) den Wert 1.

5. (*) Lösen Sie im Programm SAMP25 die quadratische Gleichung $a*x^2+b*x+c=0$.

Hinweis:

Berücksichtigen Sie zunächst den Fall $a=0$.

Testen Sie danach die Diskriminante $D=b^2-4ac$ gegen Null

6. Berechnen Sie im Programm SAMP26 für die Matrix rMatrix(3,3) mit $a(i,j)=i*j$ folgende Werte:

- a) das größte Element
- b) die Anzahl der positiven Elemente
- c) die Summe der negativen Elemente
- d) die größte Zeilensumme
- e) das betragsgrößte Element in jeder Zeile

7. (*) Berechnen Sie im Programm SAMP27 die Binomialkoeffizienten für $n=1, \dots, 10$.

Hinweis:

$B(1,1)=1$, $B(n,1)=B(1,n)=1$, $B(n+1,k)=B(n,k-1)+B(n,k)$ für $1 < k < n$.

$$\begin{array}{ccccc} & & 1 & & \\ & & 1 & 2 & 1 \\ & 1 & 3 & 3 & 1 \end{array}$$

8. (*) Berechnen Sie im Programm SAMP28 die Quadratwurzel einer interaktiv eingelesenen positiven Zahl r1 mit Hilfe folgender Iterationsformel (Newton-Formel für $x^2-r1=0$):

$$\begin{aligned} x(n) &= 1 \\ x(n+1) &= (1.0/2.0)*[(x(n)+r1/x(n)] \end{aligned}$$

Brechen Sie die Iteration ab, wenn sich zwei aufeinanderfolgende Approximationen um weniger als $EPS=1.0E-6$ unterscheiden. Vergleichen Sie mit dem Wert, den die intrinsische Funktion SQRT(r1) zurückliefert.

9. (*) Berechnen Sie beliebige positive Wurzeln analog zu Aufgabe 8.

Hinweis:

Newton-Formel: $x(n+1)=x(n)-f(x(n))/f'(x(n))$, $f(x)=x^m-r1$, $m=1,2,3,4,\dots$

12.6. Übung 6: Formelfunktionen und Funktionen

1. Schreiben und testen Sie im Programm SAMP29 Formelfunktionen für folgende Ausdrücke in den formalen Parametern x, y und z (reelle Zahlen) und n und m (natürliche Zahlen):

- a) $x^2 + y^2 + z^2$
- b) $x^2 + 2xy + y^2$
- c) $(1/2a)(-b + (b^2 - 4ac)^{0.5})$
- d) $4 \tan^{-1}(x)$
- e) $\sqrt{n+m}$

2. Schreiben und testen Sie Funktionen im Programm SAMP30, die folgende Funktionswerte berechnen:

- a) Abstand zwischen zwei Punkten im Raum
- b) Wurfweite bei gegebenem Abwurfwinkel a und gegebener Abwurfgeschwindigkeit V_0 .
- c) Mittelwert eines Vektors

Hinweis:

$W = V_0 \sin(a) \cos(a) / g$. Welches ist der optimale Abwurfwinkel a? (ggfs. Plot!)

3. (*) Schreiben Sie eine Funktion, die zu einem gegebenen Datum d/m/y den zugehörigen Wochentag berechnet und testen Sie mit dem aktuellen Datum.

Hinweis:

Verwenden Sie folgenden Algorithmus, in dem eckige Klammern den ganzzahligen Wert nach Abschneiden bezeichnen, z.B. ist $[5/2] = [2.5] = 2$:

Für $m=1$ oder $m=2$, addiere 12 zu m und subtrahiere 1 von y.
 $r = \text{MOD}(d + 2m + 2 + [(3m+3)/5] + y + [y/4] - [y/100] + [y/400], 7)$
 $r=0$ entspricht Samstag, $r=1$ entspricht Sonntag, ..., $r=6$ entspricht Freitag

4. Schreiben Sie eine Funktion, die die Fibonacci-Zahlen berechnet und testen Sie die Funktion im Programm SAMP31.

Hinweis:

$\text{Fib}(0) = \text{Fib}(1) = 1$, $\text{Fib}(n+2) = \text{Fib}(n) + \text{Fib}(n+1)$

5. (*) Schreiben Sie eine Funktion, die ein Kalenderblatt für einen gegebenen Monat ausgibt. Testen Sie im Programm SAMP32 mit dem aktuellen Monat.

6. (*) Schreiben Sie eine Funktion, die für 2 natürliche Zahlen die größte gemeinsame Primzahl berechnet. Testen Sie im Programm SAMP33 mit den Zahlenpaaren (10,5), (1,1) und (7,5).

12.7. Übung 7: Unterprogramme und COMMON Blöcke

1. Schreiben Sie ein Unterprogramm DIAG(), das eine beliebige Diagonalmatrix rA(n,n) mit einem Vektor rDiag(n) initialisiert. Testen Sie im Programm SAMP34 mit rDiag(4) /1,1,1,1/ und rDiag(5) /1,2,3,4,5/.

2. Schreiben Sie ein Unterprogramm SWAP(r1,r2), das die Werte zweier Variablen vertauscht. Testen Sie im Programm SAMP35 mit r1=5 und r2=3. Was passiert, wenn Sie Ausdrücke wie 5+3 und 6 an das Unterprogramm übergeben? Was passiert, wenn Sie einen Ausdruck und eine Variable übergeben?

3. Transponieren Sie in einem Unterprogramm TRANSP() eine Matrix in einem COMMON Block. Welchen Nachteil hat dieses Vorgehen? Testen Sie im Programm SAMP36 mit folgender Matrix iA(2,2) /1,2,3,4/.

4. Schreiben Sie ein Unterprogramm SORT(), das ein unsortiertes Feld beliebiger Größe sortiert zurückliefert. Testen Sie im Programm SAMP37 mit dem Feld iA(5) /1,3,2,4,5/.

5. (*) Schreiben Sie ein Unterprogramm MATMUL, die zwei beliebige Matrizen A(m,n) und B(n,k) miteinander multipliziert und in C(m,k) zurückliefert. Testen Sie im Programm SAMP38 mit zulässigen und unzulässigen Paaren von Matrizen. Im Falle unzulässiger Matrizen soll in der Statusvariablen iFail ein Wert -1 für einen Fehler zurückgeliefert werden:

```

SUBROUTINE MATMUL(rA,iN1,iM1,rB,iN2,iM2,rC,iN3,iM3,iFail)
REAL rA(iM1,iN1), rB(iM2,iN2), rC(iM3,iN3)
INTEGER iM1,iN1,iM2,iN2,iM3,iN3
INTEGER iFail
IFail=0
...
RETURN
END
    
```

Hinweis:

Beachten Sie, daß in FORTRAN Matrizen **spaltenweise** gespeichert werden, d.h. der linke Index läuft bei einer Anfangswertzuweisung zuerst.

$C(j,i)=A(k,i)*B(j,k)$, $k=1,\dots,iM1$, $i=1,\dots,iN1$, $j=1,\dots,iM2$,
 $iN1=iM3$, $iM1=iN2$, $iM2=iM3$

6. (*) Schreiben Sie ein Unterprogramm zur Berechnung der Gershgorin-Kreise einer Matrix A(n,n) und plotten Sie die Kreise mit einem verfügbaren Grafikpaket. Testen Sie im Programm SAMP39 mit Diagonalmatrizen und ergänzen Sie im Plot die tatsächlichen Eigenwerte!

Hinweis:

$G(i)=\{x: |x-a_{ii}| \leq \sum |a_{ik}|, k=1,\dots,n, k \neq i\}$, $i=1,\dots,n$

Die Vereinigung aller Kreise G(i) enthält alle Eigenwerte der Matrix A(n,n).

12.8. Übung 8: Ein- und Ausgabe und Dateiverarbeitung

1. Geben Sie die Variable `i1` mit dem Wert (-55) im Programm SAMP40 unter Kontrolle folgender Dateifeldbeschreibungen aus:

I5, I3,I2, I10.

2. Geben Sie im Programm SAMP41 die Variable `r1` mit dem Wert (-55.0) unter Kontrolle folgender Dateifeldbeschreibungen aus:

E10.5, E10.5E3, F3.1, F10.2, G10.2, E3.4.

3. Geben Sie im Programm SAMP42 eine ansprechende Tabelle der ersten 20 Primzahlen am Bildschirm aus.

4. Schreiben Sie die Tabelle aus Aufgabe 3 in die Datei PRIMZAHL DATA A.

5. Lesen Sie eine vorher unbekannte Anzahl von Meßwert-Tripeln (`i,r1(i), r2(i)`) aus einer Datei ein und berechnen Sie anschließend deren Anzahl, Mittelwert und Standardabweichung im Programm SAMP43.

6. Was passiert, wenn Sie bei einer WRITE oder READ Anweisung keine Formatbeschreibung `FMT=...` angeben? Versuchen Sie, die Ausgabe zu interpretieren!

7. (*) Schreiben Sie ein Prozedur `GETINP()`, die eine Datei öffnet, ggfs. eine Fehlerbehandlung durchführt, alle Datensätze in ein genügend großes Feld (z.B. unter VS FORTRAN und VM/CMS durch die Compiler-Option `DYNAMIC COMMON` definierbar) einliest und dann die geöffnete Datei wieder schließt. Versuchen Sie, die Prozedur möglichst generisch zu schreiben. Testen Sie mit Datensätzen, die Meßpunkten entsprechen und mit Datensätzen, die einer Matrix entsprechen.

8. (*) Schreiben Sie eine Prozedur `EULER` zur numerischen Lösung von Differentialgleichungen der Form $y'(x)=f(x,y)$ mit dem Eulerschen Polygonzugverfahren und testen Sie im Programm SAMP44 mit $f(x,y)=\exp(x)$. Plotten Sie die numerische und die exakte Lösung!

13. Anhang A: Wichtige CMS Kommandos

An- und Abmelden:

```
LOGIN: <username>
PASSWORD: <password>
...
LOGOFF
```

Dateiverarbeitung:

Dateiname:

```
filename filetype filemode (kurz: fn ft fm),
z.B.: SAMP1 FORTRAN A
```

COPYFILE fn1 ft1 fm1 fn2 ft2 fm2	kopiert eine Datei f1 ... nach f2 ...
ERASE fn ft fm	löscht eine Datei f1 ...
RENAME fn1 ft1 fm1 fn2 ft2 fm2	benennt eine Datei f1 ... um in f2 ...
FILELIST	ruft den Dateimanager auf

FORTRAN Entwicklungsumgebung:

XEDIT fn ft [fm]	ruft den Editor auf
FORTVS2 fn	ruft den Compiler auf
LOAD fn	ruft den Programm-Lader auf
START *	startet das geladene Programm
START * (DEBUG)	startet den FORTRAN Debugger

Drucken:

```
PRINT fn ft [fm]
```

REXX Skript zur Programmentwicklung:

```
/* REXX */
f - edits, compiles, loads and starts a
  FORTRAN program.
F. Elsner, RZUNIOS, 04.07.92
*/

arg fn
if fn='' then
do
say 'Syntax:          F <filename>'
say 'e.g.:           F PROGRAM1'

/* Edit */
say 'Starting editor ... '
'xedit' fn 'fortran a'

/* Compile */
say 'Starting compiler ...'
'fortvs2' fn

/* Load and go */
if rc = 0 then
do
say 'Loading and starting program ...'
'load' fn
'start *' /* start * (debug) for debug mode */
end
end
```