FORTRAN 77 — Die Ultrakurzanleitung

Ralf Napiwotzki
Dr.-Remeis-Sternwarte
Friedrich-Alexander-Universität Erlangen-Nürnberg
napiwotzki@sternwarte.uni-erlangen.de

12. Nov. 99

1 Programmentwicklung

Kriterien für ein gutes Programm (nicht immer sind alle vier Kriterien gleichzeitig perfekt erfüllbar, aber man sollte es wenigstens versuchen):

1. **effizient** (wenig Rechenzeit und Kernspeicher)
2. **lesbar** (Kommentare, sinnvolle Namen)
3. **transportabel** (Standard–FORTRAN, rechnerunabhängig)
4. **allgemein** (modularer Aufbau, Unterprogramme universell einsetzbar)

2 Fundamentale FORTRAN 77-Elemente

2.1 Segmente

Das Programm besteht aus Segmenten (Modulen). Der Compiler behandelt jedes Segment getrennt. Der Loader verbindet die Segmente (und evtl. benutzte Bibliotheksroutinen) zu einem lauffähigen Programm.

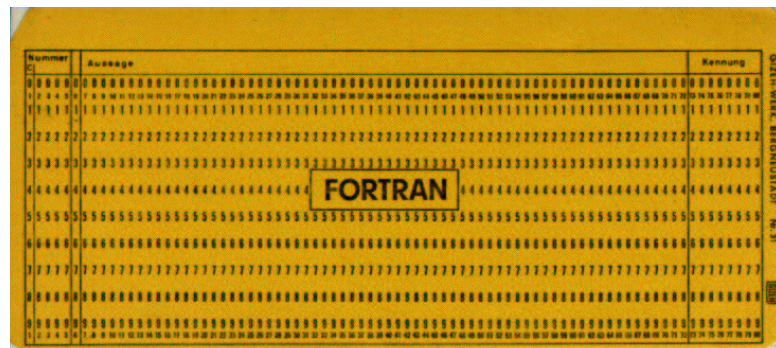
- Hauptprogramm (`PROGRAM`)
- Unterprogramme (`SUBROUTINE`)
- Funktionen (`FUNCTION`)
- COMMON–Blöcke (`COMMON / /`)
- [BLOCK DATA – Bereiche] (`BLOCK DATA`)

Jedes Segment besteht aus:

- Deklarationsteil (nichtausführbare *statements*)
- Rumpf (ausführbare *statements*)
- END (außer bei COMMON)

2.2 FORTRAN 77-Statements

Die Struktur eines FORTRAN 77-Programms geht noch auf die ursprünglich verwendeten Lochkarten zurück. Die Spalten sind jeweils für bestimmte Zwecke reserviert. In FORTRAN 90 ist *alternativ* ein freies Format zulässig.



Spaltenbelegung eines FORTRAN 77-Programms:

Spalte	Belegung
1–5	Label `C` oder `*` in Spalte 1 leitet eine Kommentarzeile ein
6	Fortsetzungskarte (max. 19)
7–72	eigentliche Statements
73–80	Kennung (wird nicht vom Compiler bearbeitet). Einige Compiler bieten die Option auch die Zeilen 73–80 oder gar 73-132 für die eigentlichen Statements zu verwenden. Aus Portabilitätsgründen ist aber davon abzuraten.

2.3 FORTRAN – Zeichenvorrat

- **Groß**buchstaben: `A-Z`
- Ziffern: 0 . . . 9

- ␣ (Leerstelle, *blank*)
- Sonderzeichen: `+ - * / ( ) , . $ ' : =`

Andere Zeichen dürfen in Standard–FORTRAN–Statements nicht vorkommen (mit der Ausnahme von *Strings*, z.B. für Filenamen oder Text)!

2.4 Namenskonvention für Segmentnamen

- maximal **6** black Zeichen
- keine Sonderzeichen
- erstes Zeichen = Buchstabe

Die meisten Compiler sind nicht mehr auf diese restriktiven Normen fixiert. ⇒ Im Zweifelsfall kann man daher der Übersichtlichkeit den Vorzug geben. FORTRAN 90 erlaubt Namen mit bis zu 31 alphanumerischen Zeichen (inkl. dem Unterstreichungszeichen `_`).

3 Variablen und Konstanten

Variable bzw. Konstanten werden durch symbolische Namen angesprochen. Die Namensgebung folgt der oben angebenen Konvention für Segmentnamen.
Richtig: `XYZ`, `ALPHA1`, `ZEHN`
Falsch: `0815`, `A$`, `HUNDERT`,

Variablennamen sind symbolische Namen für Speicherplätze; sie werden vom Compiler freigehalten und erst vom Loader oder bei Programmausführung belegt. Speicherplätze für Konstanten werden bereits beim Compilieren belegt (und sind nicht änderbar!).

3.1 Typen von Variablen bzw. Konstanten

Je nach Verwendungszweck können in FORTRAN fünf verschieden Typen verwendet werden.

ganze Zahlen: `INTEGER`, z.B. 2

reelle Zahlen: `REAL`, z.B. 2.0, 2., 3.E+17 (Punkt beachten!)

doppeltgenaue reelle Zahlen: `DOUBLE PRECISION`, z.B. 2.D0) (manchmal verwendet man auch: `REAL*4` oder `REAL*8`)

komplexe Zahlen: `COMPLEX`, z.B. (1.0,2.0) entspricht 1+2i

Text (*strings*) : `CHARACTER`, z.B. ('guten Tag') (`CHARACTER*40` bedeutet ein String mit 40 Zeichen)

logische Variable: `LOGICAL`, diese können die Werte `.TRUE.` und `.FALSE.` annehmen

In FORTRAN kann der Typ *implizit* oder *explizit* vereinbart werden. Legt man den Typ einer Variablen nicht explizit fest, gilt defaultmäßig die implizite Vereinbarung.

Implizite Typ–Vereinbarung: alle Variablen bzw. Konstanten, deren Name mit `I,J,K,L,M,N` beginnt, sind vom Typ INTEGER, alle anderen vom Typ REAL. Dieses Verfahren ist sehr fehlerträchtig und deshalb sollte die implizite Typ–Vereinbarung sollte stets ausgeschaltet werden mit `IMPLICIT NONE` (nicht Standard!). Variablen bzw. Konstanten sollten *explizit* deklariert werden. Durch `IMPLICIT NONE` erreicht man, daß die Verwendung einer undeklarierten Variablen zu einer Fehlermeldung führt.
Beispiel für den Deklarationsteil eines Programms:

```
␣␣␣␣␣␣PROGRAM␣PROG1
␣␣␣␣␣␣IMPLICIT␣NONE
␣␣␣␣␣␣INTEGER␣C
␣␣␣␣␣␣REAL␣IX,X,Y,Z
␣␣␣␣␣␣CHARACTER*10␣NAME,VORNAM*5
␣␣␣␣␣␣LOGICAL␣A
␣␣␣␣␣␣DOUBLE␣PRECISION␣B
          ⋮

␣␣␣␣␣␣END
```

3.2 Zuweisung von Werten an Variable bzw. Konstanten

Der Wert von Variablen kann während des Programmablaufs oder vorher festgelegt werden. Ist einer Variablen noch kein Wert zugewiesen worden, so kann diese – je nach Compiler – mit dem Wert `0` belegt sein. Aber Vorsicht: verläßt man sich darauf, so kann bei dem Wechsel zu einem anderen Compiler die Vorbesetzung völlig anders sein.

1. Speicherinhalt einer Variablen wird während der Ausführung des Programms belegt bzw. verändert:
   - explizite Zuweisung: `X = 0.`
   - I/O–statement: `READ(3,*) X`
   - (Einlesen von Tastatur (*stdin*): `READ *,X`)
   - Übergabeargument in Unterprogrammaufruf
   - *internal file*
   - Änderung des Speicherplatzinhalts über COMMON–Block–Zugriff oder EQUIVALENCE–Anweisung
   - Änderung des Speicherplatzinhalts durch freien Zugriff auf (beliebige) Array–Elemente
2. Loader besetzt über die nicht–ausführbare Anweisung `DATA` die Speicherplätze einer Variablen vor: `DATA X,Y,Z /3*0./` (im Deklarationsteil). Bei Verwendung dieser Methode kann der Wert während des Programmablaufs geändert werden.

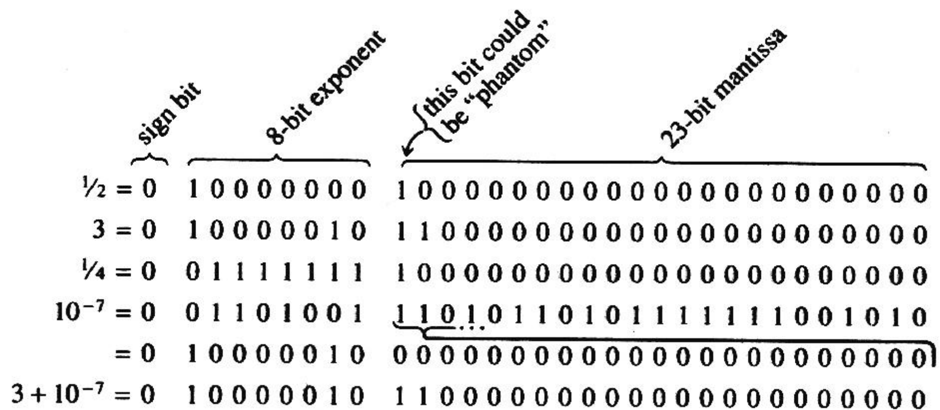
3. Compiler legt Wert einer Konstanten fest: `PARAMETER (PI=3.1415D0)` (im Deklarationsteil). Bei Verwendung dieser Methode ist der Wert nicht nachträglich änderbar.

3.3 Zahlendarstellung

Für die Darstellung numerischer Variablen und Konstanten steht naturgemäß nur ein beschränkter Speicherbereich zur Verfügung. Damit sind darstellbarer Zahlenbereich und/oder Genauigkeit limitiert. INTEGER- und REAL-Zahlen belegen i.d.R. 4 Bytes (= 32 Bits) und DOUBLE-PRECISION-Zahlen 8 Bytes.

Ganze Zahlen (INTEGER): exakte Arithmetik (sofern im darstellbaren Zahlenbereich). Mit 4 Bytes ist der Zahlenbereich auf den Bereich $-2^{31} < i < 2^{31}$ beschränkt.

Gleitkommazahlen (REAL, DOUBLE PRECISION): endliche Darstellungsgenauigkeit $\Rightarrow$ numerischer Rundungsfehler!!



- Zahlendarstellung:

$$s \times M \times B^{e-E}$$

  (s: Vorzeichen, M: Mantisse, B: Basis (typ. 2 oder 16), e: Exponent, E: *bias* (maschinenabhängig))

- *normalized number*: „linkes" Bit in Mantisse = 1

3.3.1 Maschinengenauigkeit und Fehler

Maschinengenauigkeit: ϵ_m: Die kleinste Gleitkommazahl, die zu 1 addiert eine – in Maschinendarstellung – von 1 abweichende Zahl ergibt (REAL (4 Byte): typ. $3 \cdot 10^{-8}$).

Rundungsfehler durch endlich genaue Zahlendarstellung. Rundungsfehler nach N Operationen: $\sim \sqrt{N}\epsilon_m$ (falls die einzelnen Fehler zufällig verteilt sind!) Der Rundungsfehler kann erheblich größer werden, etwa bei Subtraktion ähnlich großer Zahlen; durchaus häufiges Auftreten solcher Probleme:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Abbruchfehler: (hervorgerufen durch diskrete Näherung vieler numerischer Größen, etwa endliche Summation statt einer unendlichen Taylorreihe etc.).

Abbruchfehler können – im Prinzip – vom Programmierer umgangen bzw. minimiert werden, Rundungsfehler sind dagegen nicht vermeidbar und lediglich durch geeignete Methoden bzw. Typdeklarationen zu reduzieren. Es ist **WICHTIG** sich bei numerischen Verfahren über diese Fehlerquellen im klaren zu sein. *Formal (bei unendlicher Genauigkeit) richtige Routinen können unsinnige Ergebnisse liefern!*

4 Ein- und Ausgabe

FORTRAN behandelt die Ein- und Ausgabe von/auf Datenträger (Festplatte) und Tastatur und Bildschirm analog.

- Lesen: `READ(K,L)`
- Schreiben: `WRITE(K,L)`

wobei K = Kanalnummer, L = Label

Kanalnummer: Die Kanalnummer darf im Bereich $0 \leq K \leq 99$ liegen. Man kann von einem Programm aus verschiedene Kanäle benutzten. Einige Kanalnummern sind (compiler- und rechnerabhängig) vordefifiert. Häufig gilt:

5 : Standardeingabe (Tastatur)

6 : Standardausgabe (Bildschirm)

7 : Stanzausgabe

Kanalnummern ≥ 10 sind "sicher". Ohne Verwendung des `OPEN`-Statements (s.u.), greift Schreiben und Lesen des Kanals 10 auf die Datei fort.10 zu. Mit dem `OPEN`-Statement ist die Vergabe individueller Dateinamen möglich. Die Standardein- und ausgabe kann auch mit der Kanal"nummer" `*` angesprochen werden.

Label: Das Label dient zur Angabe des Ausgabeformats. Das kann durch Angabe eines `FORMAT`-Statements oder anders erfolgen.

- L=*: formatfreies (oder auch "listengesteuertes" Lesen bzw. Schreiben. Die Formatierung erfolgt automatisch. Beispiel:

```
␣␣␣␣␣␣WRITE(6,*)␣'guten␣Tag'
```

- L=Nummer eines `FORMAT`–Statements. Damit hat man die Möglichkeit das Ausgabeformat detailiert zu kontrollieren. Beispiel:

```
␣␣␣␣␣␣WRITE(6,100)␣'guten␣Tag'
100␣␣␣FORMAT(A)
```

Zur Erinnerung: Labels befinden sich in den ersten 5 Spalten einer Zeile.

- L=Formatangabe Beispiel:

```
␣␣␣␣␣␣WRITE(6,'(A)')␣'guten␣Tag'
```

4.1 Das FORMAT–Statement

`FORMAT` ist eine nicht–ausführbare Anweisung, die an beliebiger Stelle eines Moduls auftreten darf. Sie muß ein Label haben.

```
<Labelnummer>  FORMAT(Spezifikatorenliste)
```

Spezifikatoren

`X`	Leerstelle
`In`	Integer der Länge n
`Fn.m`	Festkommazahl mit n Stellen, wobei m Nachkommastellen vorhanden sind
`En.m`	Exponentialdarstellung
`nP`	Skalenfaktor (multipliziert Mantisse mit 10^n, während der Exponent um n verringert wird. Vorsicht bei F-FORMAT)!
`Gn.m`	F bzw. E
`An`	Text der Länge n
`A`	beliebiger Text
`/`	erzeugt neue Zeile.

Wiederholungsfaktoren möglich, z.B. `5I7` oder `5(2X,I7)`. Text darf in FORMAT–Statenment eingebunden werden, z.B. `FORMAT(1X,' x = ',F10.3)`

: Beispiele:

1. Nicht–formatiertes Schreiben in file `fort.10`

```
␣␣␣␣␣␣WRITE(10,*)␣X,Y,Z
```

2. Formatiertes Schreiben auf Standard–Output:

```
␣␣␣␣␣␣WRITE(6,111)␣X,Y,Z
111␣␣␣FORMAT(1X,'x=␣',F10.2,E10.2,/,F10.2)
```

3. Lesen von fort.11

```
␣␣␣␣␣␣READ(11,*)␣X,Y
```

4.2 Direkte Ausgabe auf Standard–Output

```
PRINT <Formatlabel> <Ausgabeliste>
```

Beispiel:

```
PRINT 10,A,B,C
PRINT*,A,B,C
```

5 Wertzuweisungen

5.1 Arithmetische Ausdrücke

Eine Wertzuweisung wird wie eine mathematische Formel geschrieben: verb-X=(Y+Z)/3.-. Dabei erfolgt die Zuweisung von rechts nach links: `I=I+1`

Zur Verfügung stehen neben den Grundrechenarten `+`, `-`, `*`, `/` auch zahlreiche weitere Operationen. Z.B.

- Potenzieren: `**`, z.B. verb+A**2.5+
- Wurzel: `SQRT(X)`
- Exponentialfunktion: `EXP(X)`

Man achte darauf nur Variablen vom selben Typ zu verwenden, um unnötige Typumwandlungen zu vermeiden.

5.2 Logische (boolesche) Ausdrücke

Neben arithmetischen kann man auch mit logischen Ausdrücken arbeiten. Es stehen die Operatoren `.AND.`, `.OR.` und `.NOT.` zur Verfügung (weniger wichtig die Äquivalenzoperatoren `.EQV.`, `.NEQV.`). Die Punkte gehören zum Namen! Die Zuweisungen erfolgen wie bei arithmetischem Ausdrücken:

```
LOGICAL A,B,C
   ⋮
A = B .AND. C
```

Es stehen Vergleichsoperatoren für größer (gleich), Gleichheit, Ungleichheit und kleiner (gleich) zur Verfügung: `.LT. .LE. .EQ. .NE. .GT. .GE.`

```
REAL B,C
LOGICAL A

   ⋮
A = B .LT. C
```

Die Vergleichsoperatoren funktionieren auch mit CHARACTER–Variablen (Vergleichskriterium: Alphabet):

```
CHARACTER*10 NAME
LOGICAL A
```

```
    ⋮
B = NAME .LT. 'D'
```

Insbesondere bei Sonderzeichen hängt das Ergebniß aber vom maschineninternen Zeichensatz ab. Deshalb gibt es alternativ lexikalische Vergleichsfunktionen für CHARACTER-Variablen: `LLT`, `LLE`, `LGT`, `LGE`

```
B = LLT (NAME,'D')
```

6 Programmsteuerung

6.1 Mittel zur Programmverzweigung:

6.1.1 GOTO–Anweisung

Mit GOTO-Anweisungen springt man zu einem Label. Mit GOTOs sollte man sparsam umgehen, weil großzügiger Gebrauch die Programmstruktur sehr unübersichtlich machen kann. Nebenbei hinderen zu viele GOTOs moderne Compiler unter Umständen daran effizienten Code zu produzieren.

Unbedingtes GOTO: `GOTO L` (L=Labelnummer)

Bedingtes GOTO: `GOTO(L1,L2,...) I` (L1,L2,...=Labelnummer, I: Integervariable, die angibt, welche Labelnummer verwendet wird). Finger weg!!!!

6.1.2 IF–Anweisung:

Mit IF-Anweisungen kann man Fallunterscheidungen durchführen. Es gibt drei Versionen.

Logisches IF: `IF (<logischer Ausdruck>) <Statement>` (Statement kann jede ausführbare Anweisung sein außer `IF, DO, ELSE, ELSE IF, ENDIF, END`), z.B.:

```
IF(A.GT.10) B=3.0
```

Arithmetisches IF: `IF(<arithmetischer Ausdruck>) L1,L2,L3`. Antik! Finger weg!!!

Block–IF–Anweisung: Die Block-IF-Anweisung erlaubt die Konstruktion auch von komplexen Fallunterscheidungen und ist in vielen Fällen dem Einsatz von GOTOs überlegen.

einfachste Form:

```
IF(<logischer Ausdruck>) THEN
   ⋮
ENDIF
```

ELSE–Anweisung:

```
IF(<logischer Ausdruck>) THEN
   ⋮
ELSE
```

```
   ⋮
ENDIF
```

Verschachtelung mit `ELSE` oder `ELSE IF`, z.B.:

```
IF(<logischer Ausdruck>) THEN
   ⋮
ELSE
  IF(<logischer Ausdruck>) THEN
      ⋮
  ELSE
      ⋮
  ENDIF
ENDIF
```

oder `ELSE IF`-Konstruktion

```
IF(<logischer Ausdruck>) THEN
   ⋮
ELSE IF (<logischer Ausdruck>) THEN
   ⋮
ELSE
   ⋮
ENDIF
```

- nur *ineinander* schachteln!
- Einrückungen für Übersichtlichkeit

Beispiel:

mit geschachtelten `IF`-Schleifen

```
IF(A.GT.10) THEN
   X = 3.*Y + 2.
ELSE
   IF(B.GT.0.) THEN
      Z = 3.5
      X = 2.0
   ELSE
      X = 3.0
   ENDIF
ENDIF
```

oder als ELSE IF–Konstruktion

```
IF(A.GT.10) THEN
   X = 3.*Y + 2.
ELSE IF(B.GT.0.) THEN
      Z = 3.5
      X = 2.0
ELSE
      X = 3.0
ENDIF
```

6.2 DO–Schleifen

FORTRAN 77-Form der DO–Schleife:

```
DO L I=M1,M2,M3
```

(L=Labelnummer, I=Schleifenindex, M1=Startwert, M2=Endwert, M3=Schrittweite (Default=1)). `I,M1,M2,M3` sollen/dürfen in der Schleife nicht verändert werden.

Die letzte Anweisung muß ausführbar sein, d.h. nicht: `DO`, `GOTO`, `PAUSE`, `RETURN`, arithm. `IF`, Block–`IF`, `ELSE IF`, `ELSE`, `ENDIF`, `END`. In der Schleife darf kein `RETURN` auftreten.

Empfehlungen: 1) `CONTINUE` als letzte Anweisung einer Schleife, 2) INTEGER–Variable als Schleifenindex (reelle Zahlen möglich, aber *loop*–Länge hängt von interner Rechengenauigkeit ab!)

Beispiel:

```
      DO 100 I=1,15
      J = 3 + I
         ⋮
100   CONTINUE
```

Alternative Form der DO-Schleife: Abschluß mit `END DO`. *Kein Standard in FORTRAN77, erst in FORTRAN90 Standard!*

```
      DO I=M1,M2,M3
         ⋮
      END DO
```

Beispiel:

```
      DO I=1,15
      J = 3 + I
         ⋮
      END DO
```

6.2.1 Nützliche Iterationen

In anderen Programmiersprachen wie PASCAL gibt es nützliche Konstrukte wie DO–WHILE– und DO–UNTIL–Iterationen, die in FORTRAN77 nicht zur Verfügung stehen, daher:

DO–WHILE in FORTRAN77:

```
      N = 1
100   IF(N.LT.1000) THEN
         N = 2 * N
         J = J + 1
         GOTO 100
      ENDIF
```

DO–UNTIL in FORTRAN77:

```
      N = 256
100   CONTINUE
         N = N / 2
         J = J + 1
      IF(N.NE.1) GOTO 100
```

Initialisierung nicht vergessen (⇒ Endlos–Schleife!)

7 Kommentare

Kommentare beginnen in der ersten Spalte:

```
C       Dies ist ein Kommentar
```

Beim Kommentieren sollte man folgendes beachten:

- Übersichtliche, ausgewogene und strukturierte Kommentare
- ... im *header* eines Programmsegments:
  Zweck der Routine, Hinweis auf Methode/Algorithmus, Ein– und Ausgabeparameter, Besonderheiten, Autoren bzw. Entwicklungs'geschichte', Version, Datum
- ... im Rumpfteil Gliederung logisch gekoppelter Abschnitte, algorithmische Hinweise, Unterprogrammaufrufe, komplizierte Funktionen, nicht Überkommentieren!

8 Indizierte Variable (Felder)

Vektoren, Matrizen o. ä. können mit Feldern (*array*) dargestellt werden. Deklaration:

```
REAL A
DIMENSION A(N)
```

Besser:

```
REAL A(N)
```

(Indexbereich von 1 bis N). Oder

```
REAL A(M:N)
```

(M. . .N: Bereich 'erlaubter' Indices) Der Index–Bereich darf negativ sein.

Bis zu 7 Indices sind erlaubt, allerdings ist es meist zu empfehlen sich auf nicht mehr als 3 Indices zu beschränken.

```
REAL A(M,N)
```

(1. Index von 1 bis M, 2. Index von 1 bis N)

Die Darstellung im Speicher: eindimensional, seriell. Der am weitesten links stehende Index läuft am schnellsten bei C gerade umgekehrt!). Indices dürfen aus einfachen arithmetischen Berechnungen (`*,+,-`) bestehen. **Das Überschreiten der Indexgrenzen führt normalerweise zu keiner Fehlermeldung!** *Rightarrow* Kontrolle durch Programm(iererIn)! (aber: sog. *array–bound–checking* häufig als Compiler–Option vorhanden)

In FORTRAN 77 müssen die Feldelemente einzeln angesprochen werden. Vektornotation (z.B. A=0.) ist nicht moeglich.

Beispiele:

1. `REAL A(6),B(-3:5),C(-1:1,2:5)`

2. ```
   PARAMETER(NMAX=1000)
   REAL A(NMAX),B(NMAX,NMAX+1)
   ```

3. `X = A(5*I3)`

9 Programmsegmente

Programmsegmente können nacheinander in einem File stehen oder auf mehrere Files verteilt werden, die vom Compiler einzeln bearbeitet werden.

9.1 Hauptprogramm

```
PROGRAM name
  ⋮
END
```

9.2 Unterprogramm

```
SUBROUTINE name (<Parameterliste>)
<Deklarationsteil>
<Anweisungsteil>
  ⋮
RETURN
END
```

Die Parameterliste darf leer sein.
Beispiel:

```
SUBROUTINE UNTER(X,Y,Z)
```

oder

```
SUBROUTINE SUB1
```

Aufruf des Unterprogramms:

```
CALL <Unterprogrammname>(<Parameterliste>)
```

z.B.

```
CALL UNTER(A,B,C)
```

oder

```
CALL SUB1
```

Ein Unterprogramm darf weitere Unterprogramme aufrufen. Allerdings sind rekursiven Aufrufe nicht erlaubt (in FORTRAN90 gibt es die Möglichkeit rekursive Funktionen zu definieren). In verschiedenen Segmenten dürfen gleiche Namen für unterschiedliche Variable benutzt werden (lokale Variable). Weiterreichen von Variablen erfolgt entweder durch Parameterliste oder über COMMON–Block.
```

Beispiel:

```
        ⋮
      A = 5.5
      B = 3.2
      CALL SUB(A,B,C)
        ⋮
      END
C**********************************
      SUBROUTINE SUB(X,Y,Z)
      REAL X,Y,Z
      Z = X * Y
      END
```

A,B,C sind *aktuelle Parameter*. **X,Y,Z** sind *formale Parameter*, d.h. sie belegen keinen Speicherplatz.

9.2.1 Übergabeparameter

Variable werden im *call–by–reference*–Verfahren übergeben, d.h. es wird nur die Speicheradresse, nicht aber der Variablenwert weitergegeben (sog. *call–by–value*). Übergeben werden dürfen Variable und Konstanten; letztere dürfen aber **KEINESFALLS** im Unterprogramm verändert werden (wird nicht vom Compiler überprüft!). Auch Zahlen dürfen direkt in der Parameterliste übergeben werden (⇒ Konstante).

FORTRAN überprüft nicht, ob die übergebenen Variablen den im Unterprogramm deklarierten und erwarteten Variablentyp entsprechen. ⇒ **ProgrammiererIn muß selbst Konsistenz der Übergabeliste sicherstellen!** (häufigste ernsthafte Fehlerquelle, kryptische Ergebnisse!) Empfehlung: keine Zahlen in CALL–Aufruf, explizites Prüfen der Variablenliste auf Konsistenz

9.2.2 Felder als Übergabeparameter

Problem: Übergabe eines Arrays als Parameter an eine Subroutine. Wie erhält man eine konsistente Dimensionierung des Feldes?

Dimension mitübergeben:

```
      PROGRAM PROG
      DIMENSION A(100)
        ⋮
      N = 100
      CALL SUB1(A,N)
        ⋮
      C**************************************
      SUBROUTINE SUB1(X,M)
      DIMENSION X(M)
        ⋮
```

```
END
```

Dimension "kennen":

```
SUBROUTINE SUB1(X)
DIMENSION X(100)
  ⋮
END
```

"Feld mit übernommener Länge":

```
SUBROUTINE SUB1(X)
DIMENSION X(*)
  ⋮
END
```

COMMON–Block bzw. INCLUDE–File:

```
SUBROUTINE SUB1(X)
COMMON /PARA/ N
DIMENSION X(N)
  ⋮
END
```

9.3 Funktion

```
<Funktionstyp> FUNCTION name(<Parameterliste>)
  ⋮
name = ...
END
```

Bei Funktion darf die Parameterliste nicht leer sein. Man kann aber ein *dummy*–Argument darf aber übergeben werden. Parameterübergabe wie bei den SUBROUTINEn *(call–by–reference)*.

Die erlaubten Funktionstypen entsprechen den Variablentypen: REAL, INTEGER, LOGICAL, DOUBLE PRECISION, CHARACTER, COMPLEX (ansonsten gilt die implizite Typvereinbarung). Dem Funktionsnamen **muß** innerhalb der Funktion ein Wert zugewiesen werden. Benutzerdefinierte Funktionen müssen im rufenden Programm deklariert werden.
Beispiel:

```
DOUBLE PRECISION FUNCTION F1(X)
DOUBLE PRECISION X
F1 = X**2
END
```

9.3.1 vordefinierte (intrinsische) Funktionen

- Wurzel: `SQRT(X)`

- Exponentialfunktion: `EXP(X)`
- Betrag: `ABS(X)`
- Vorzeichen: `SIGN(X,Y) = sgn(Y) * |X|`
- Maximum/Minimum: `MAX(X,Y), MIN(X,Y)`
- ganzzahliger Rest bei Division: `MOD(X,Y)`
- natürlicher Logarithmus: `LOG(X)`
- dekadischer Logarithmus: `LOG10(X)`
- trigonometrische Funktionen, z.B. `SIN(X)`
- zugehörige Umkehrfunktionen, z.B. `ACOS(X)`. Besonderheit: auch 'quadrantengerechter' Arcustangens: `ATAN2(Y,X)`.
- Hyperbelfunktionen, z.B. `SINH(X)` (aber nicht die zugehörigen Umkehrfunktionen!).
- Typkonversion: `REAL(I)`, `DBLE(I)`, `INT(X)`, `NINT(X)`,...
- ...
- + maschinen–abhängige Funktionen (z.B. CPU–Zeit, Datum, weitere Funktionen etc.)

9.3.2 Funktionen als Übergabeparameter

FORTRAN gestattet auch, Funktionen als Übergabeparameter zu behandeln, sofern sie mit

```
      EXTERNAL <Funktionsname>
```

im rufenden Programm deklariert wurden. Auch intrinsische Funktionen (z.B. `SIN(X)`) können übergeben werden, wenn sie mit

```
      INTRINSIC <Funktionsname>
```

im rufenden Programm deklariert wurden. Allerdings dürfen die Typkonversionsfunktionen (z.B. `INT`), die Maximums– und Minimumsfunktion sowie die lexikalischen Funktionen (z.B. `LGE`) nicht als Parameter übergeben werden. User–definierte Funktionsnamen haben stets Vorrang vor den intrinsischen.
Beispiel:

```
      PROGRAM ZEROS
      EXTERNAL F1
        ⋮
      CALL BISEC(A,B,F1)
        ⋮
      END
C*********************************
      SUBROUTINE BISEC(X,Y,F)
      DOUBLE PRECISION X,Y,F,Y1,Y2
        ⋮
```

```
      Y1 = F(X)
      Y2 = F(Y)
        ⋮
      END
C**********************************
      DOUBLE PRECISION FUNCTION F1(Z)
        ⋮
      F1 = ...
        ⋮
      END
```

9.4 COMMON–Block

```
      COMMON /<COMMON-Block-Name>/ <Variablenliste>
```

COMMON–Blöcke gestatten gemeinsamen Zugriff unterschiedlicher Programmsegmente auf gleiche Speicherbereiche (⇒ eine weitere Variante Variablen 'auszutauschen'). Der COMMON–Block muß im Deklarationsteil eines Segments stehen. COMMON–Blöcke sind eigenständige Segmente, die im Kernspeicher hinter dem Segment stehen, in dem sie das erste Mal vorkommt. *FORTRAN überprüft nicht die Konsistenz der COMMON–Blöcke in verschiedenen Segmenten. ProgrammiererIn muß selbst Konsistenz der COMMON–Blöcke, d.h. der in ihnen definierten Variablen, sicherstellen!* ("schöne" Quelle für unauffindbare Fehler...!) Beispiel:

```
      PROGRAM PROG1
      DOUBLE PRECISION X
      COMMON /PARA/ X
      CALL SUB1
      WRITE(6,*) X
      END
C****************************************
      SUBROUTINE SUB1
      DOUBLE PRECISION X
      COMMON /PARA/ X
      X=4.D0
      END
```

Empfehlungen:

1. Bei Änderungen an einem COMMON–Block **SOFORT** alle anderen Stellen, an denen er auftritt, ebenfalls aktualisieren.

2. **NIEMALS** unterschiedliche Variablentypen verwenden oder Reihenfolge der COMMON–Block–Variablen ändern.

3. Verwendung von INCLUDE–Files

9.5 INCLUDE–Files

INCLUDE–Files können an beliebiger Stelle im Programm eingefügt werden (allerdings Nicht–Standard–FORTRAN 77/90). INCLUDE–Files enthalten typischerweise alle relevanten Deklarationen und COMMON–Blöcke, z.B.

File prog1.f

```
      PROGRAM PROG1
      INCLUDE 'prog1.inc'
      CALL SUB1
      WRITE(6,*) X
      END
C*****************************************
      SUBROUTINE SUB1
      INCLUDE 'prog1.inc'
      X=4.D0
      END
```

File prog1.inc

```
      DOUBLE PRECISION X
      COMMON /PARA/ x
```

10 Arbeiten mit Files

10.1 Dateibefehle

Öffnen von Dateien: `OPEN(<Parameterliste>)`

Parameter:

`Kanalnummer: UNIT=u`

`Filename: FILE='<Filename>'`

`Fehlerbehandlung: ERR=<Labelnummer>`

`Rekordlänge: RECL=<Rekordl"ange>`

`Status: STATUS=t`

- `'OLD'` existiert schon
- `'NEW'` neu anlegen
- `'UNKNOWN'` (Default)
- `'SCRATCH'` wird nach dem Schließen des Kanals gelöscht

FORM=a

'FORMATTED' (Default) ASCII–File

'UNFORMATTED' Binärformat

ACCESS=a

'DIRECT'

'SEQUENTIAL' (Default)

Beispiel:
```
OPEN(20,FILE='out.dat',STATUS='OLD',ERR=40)
```

Schließen von Dateien:

```
CLOSE(<Kanalnummer>)
```

Datenzeiger zum Fileanfang:

```
REWIND(<Kanalnummer>)
```

Datenzeiger um einen Eintrag zurücksetzen:

```
BACKSPACE(<Kanalnummer>)
```

Eigenschaften einer Datei bestimmen:

```
INQUIRE(FILE=<Filename>,<Parameterliste>)
```
oder
```
INQUIRE(UNIT=<Kanalnummer>,<Parameterliste>)
```
Beispiel:
```
INQUIRE(FILE='out.dat',FORM=format)}
```
format ist dann entweder 'FORMATTED' oder 'UNFORMATTED'.

10.2 Weiteres zu READ und WRITE

Unformatierte Ein- und Ausgabe:
Daten in maschineninterner Form (*binary*) schrieben bzw. lesen:
```
READ(u) <I/O-Liste>
WRITE(u) <I/O-Liste>
```
Bei jedem READ oder WRITE wird einen Record weitergegangen.
Vorteil: kompakt, d.h. weniger Speicherplatzbedarf und schnelleres Einlesen
Nachteil: die Dateien sind nicht direkt lesbar (Bildschirm/Drucker) und nicht kompatibel zu anderen Architekturen, d.h. die Daten sind auf anderen Rechnern und evt. auch von anderen Programmiersprachen aus nicht lesbar.
⇒ Anwendung bei große Datenmengen.

Parameter der READ- und WRITE-Befehle:

```
READ(u,<Steuerliste>) <I/O-Liste>       WRITE(u,<Steuerliste>) <I/O-Liste>
```

- Kanalnummer: `UNIT=u`
- Formatanweisung: `FMT=<Label>` bzw. `FMT='(a)'`
- Zugriff auf bestimmten Datensatz `r`: `REC=r` (File muß hierzu mit `ACCESS='DIRECT'` geöffnet sein.)
- Error–Parameter: `ERR=<Label>`
- Input/Output–Status: `IOSTAT=i}` (`r` gibt maschinenabhängig Auskunft über die aufgetretene Fehlerart; `i`=0: kein Fehler).

```
      READ(12,1000,IOSTAT=IP,ERR=200)
        ⋮
200   WRITE(3,'(I5)') IP
```

- Dateiende: `END=<Label>` (Programm wird bei angegebenem Label fortgesetzt, wenn *End–of–File* gelesen wurde. ⇒ Nützlich bei Lesen eines Files unbekannter Länge!

10.3 Interne Dateien

Eine interne Datei ist keine Datei im üblichen Sinne, sondern ein sequentielles File mit festgelegter maximaler Länge = CHARACTER–Variable! Die CHARACTER–Variable wird formal wie ein File beschrieben bzw. gelesen. Interne Dateien erlauben die bequeme Übertragung von numerischen Ergebnissen in Strings (inklusive Rundung!). Wichtige Anwendungen sind das Einlesen von gemischten Dateien mit Strings und Zahlen und die Umwandlung von Zahlen in geeignet formatierte String, z.B. für die Übergabe an Grafikprogramme. Man kann damit auch Dateinamen mit fortlaufender Numerierung erstellen (z.B. out.001, out.002, ...)
Beispiel:

```
        ⋮
      CHARACTER*30 OUTTXT
        ⋮
      A = 17.85
      WRITE(OUTTXT,10) A
10    FORMAT(1X,'a=',F5.1)
      WRITE(*,*) OUTTXT
        ⋮
```

ergibt:
`a= 17.9`

11 weitere FORTRAN–Statements...

`PAUSE` Unterbrechung des Programms mit Fortsetzungsmöglichkeit: `PAUSE <Text>`

`EQUIVALENCE` Zuordnung desselben Speicherplatzes zu unterschiedlichen Variablennamen: `EQUIVALENCE(x,y)`. Nützlich bei großen Hilfsfeldern unterschiedlicher Dimensionierung ⇒ Speicherplatzersparnis! **Aber:** ein sehr wirksames Hilfsmittel um Programme unübersichtlich zu machen und kryptische Ergebnisse zu erzeugen. FORTRAN 90 bietet modernere Hilfsmittel zur Speicherverwaltung.

`ENTRY` Alternativer "Einstiegspunkt" in eine SUBROUTINE: `ENTRY <Name> (Variablenliste)`. Meist ist es geschickter *innerhalb* einer SUBROUTINE eine Fallunterscheidung zu machen, z.B. mit IF... THEN... ELSE-Blöcken.