Fortran

Wikibooks

Bibliografische Information
Detaillierte Daten zu dieser Publikation sind bei Wikibooks zu erhalten:
HTTP://DE.WIKIBOOKS.ORG/
Namen von Programmen und Produkten sowie sonstige Angaben sind häufig geschützt. Da es auch freie Bezeichnungen gibt, wird das Symbol ® nicht verwendet.
Erstellt am 10. März 2011.

Diese Publikation ist entstanden bei Wikibooks, einem Projekt für Lehr-, Sach- und Fachbücher unter den Lizenzen Creative Commons Attribution/Share-Alike (CC-BY-SA) und GFDL.
PDF- und Druckversion sind entstanden mit dem Programm wb2pdf unter GPL. Dabei wurde das Textsatzprogramm LaTeX verwendet, das unter der LPPL steht.
Einzelheiten und Quellen dazu stehen im Anhang.

Inhaltsverzeichnis

1 http://de.wikibooks.org/wiki/Fortran%3A%20Fortran%2095%3A%20Standardfunktionen%3A%20Datentypfunktionen%20
2 http://de.wikibooks.org/wiki/Fortran%3A%20Fortran%2095%3A%20Standardfunktionen%3A%20Mathematische%20Funktionen%20
3 http://de.wikibooks.org/wiki/Fortran%3A%20Fortran%2095%3A%20Standardfunktionen%3A%20Stringfunktionen%20
4 http://de.wikibooks.org/wiki/Fortran%3A%20Fortran%2095%3A%20Standardfunktionen%3A%20Feldfunktionen%20
5 http://de.wikibooks.org/wiki/Fortran%3A%20Fortran%2095%3A%20Standardfunktionen%3A%20Zeigerfunktionen%20
6 http://de.wikibooks.org/wiki/Fortran%3A%20Fortran%2095%3A%20Standardfunktionen%3A%20Bitfunktionen%20
7 http://de.wikibooks.org/wiki/Fortran%3A%20Fortran%2095%3A%20Standardfunktionen%3A%20Weitere%20Funktionen%20
8 http://de.wikibooks.org/wiki/Fortran%3A%20Fortran%2095%3A%20Standardfunktionen%3A%20Intrinsische%20Subroutinen%20

1. Einleitung

FORTRAN[1]

Fortran ist eine Programmiersprache, die insbesondere für numerische Berechnungen eingesetzt wird. Der Name entstand aus *FORmula TRANslation* und wurde bis zur Version FORTRAN 77 mit Großbuchstaben geschrieben.

1.1. Geschichte

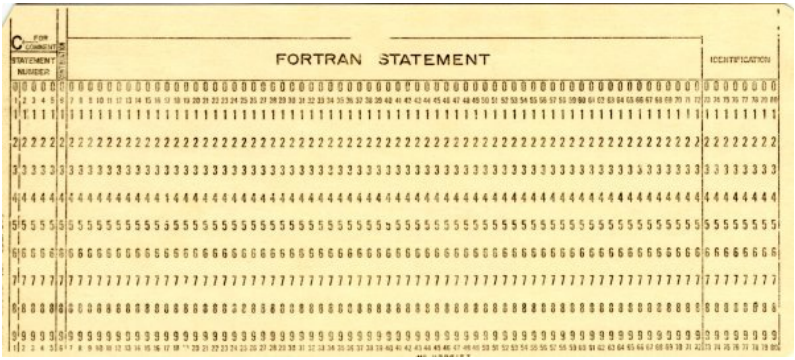


Abb. 1: Eine FORTRAN-Lochkarte aus den Anfangstagen des Computerzeitalters

1 HTTP://DE.WIKIPEDIA.ORG/WIKI/%20FORTRAN

Fortran gilt als die erste jemals tatsächlich realisierte höhere Programmiersprache. Sie geht zurück auf einen Vorschlag, den John W. Backus, Programmierer bei IBM, 1953 seinen Vorgesetzten unterbreitete.

Dem Entwurf der Sprache folgte die Entwicklung eines Compilers durch ein IBM-Team unter Leitung von John W. Backus. Das Projekt begann 1954 und war ursprünglich auf sechs Monate ausgelegt. Tatsächlich konnte Harlan Herrick, der Erfinder der später heftig kritisierten Goto-Anweisung, am 20. September 1954 das erste Fortran-Programm ausführen. Doch erst 1957 wurde der Compiler für marktreif befunden und mit jedem IBM 704-System ausgeliefert. Backus hatte darauf bestanden, den Compiler von Anfang an mit der Fähigkeit zu Optimierungen auszustatten: er sah voraus, dass sich Fortran nur dann durchsetzen würde, wenn ähnliche Ausführungsgeschwindigkeiten wie mit bisherigen Assembler-Programmen erzielt würden.

1.1.1. Versionen

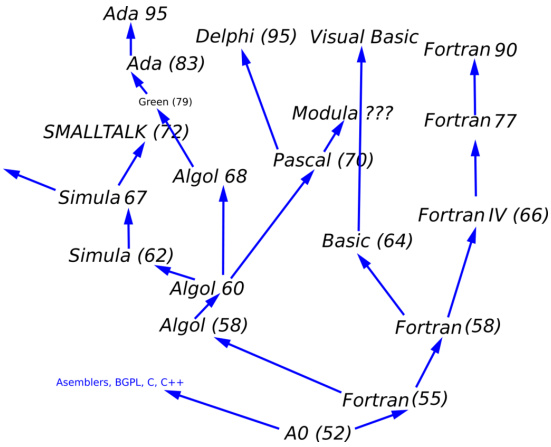


Abb. 2: Genealogie

Fortran wurde mehrmals erweitert. Viele neue Sprachelemente wurden zunächst von einem einzelnen Hersteller eingeführt und später in den internationalen Standard übernommen. Als Versionen folgten aufeinander FORTRAN I, FORTRAN II, FORTRAN IV, FORTRAN 66, FORTRAN 77, Fortran 90, Fortran 95 und zuletzt Fortran 2003. Ab FORTRAN 66 ist Fortran von einer internationalen Organisation standardisiert. Die Fortschreibung der Standards ist ein komplizierter Prozess, der oft wesentlich länger dauert als zunächst angestrebt: Der Nachfolger des 1978 erschienenen Standards FORTRAN-77, der als Fortran 8x bezeichnet wurde, war ursprünglich für das Jahr 1982 geplant, später dann für das Jahr 1985, und wurde schließlich unter der Bezeichnung Fortran90 erst am

11. April 1991 als neuer Standard und Nachfolger von FORTRAN-77 angenommen.[2]

Im Laufe dieser Erweiterungen wurden zahlreiche Sprachelemente aus neueren Programmiersprachen übernommen. Beruhte früher Fortran-Stil noch ganz auf Goto-Anweisungen, kann man seit FORTRAN 77 uneingeschränkt strukturiert programmieren. Mit Fortran 90 wurde das aus der Lochkartenzeit stammende Zeilenformat freigegeben. Ab Fortran 90 wurden interessante Elemente eingeführt, die auch z.B. in Ada vorhanden sind, beispielsweise optionale Parameter und die Möglichkeit, Prozedurparameter nicht nur über die Position in der Parameterliste zu identifizieren, sondern über ihren Namen.

1.1.2. Varianten

Einige von Fortran abgeleitete Programmiersprachen bzw. Dialekte von Fortran sind beispielsweise Ratfor, F und HPF (High Performance Fortran). Auf Fortran aufgesetzt ist das Finite-Elemente-Programmpaket Nastran.

1.2. Eigenschaften

Fortran war und ist für numerische Berechnungen vorgesehen und optimiert. Von Anfang an hatte Fortran den Potenz-Operator **. Dieser ist in vielen anderen Hochsprachen nicht vorhanden. Weiters kennt Fortran einen Datentyp für komplexe Zahlen.

2 Vorwort von Michael Metcalf in: W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery: *Numerical Recipes in Fortran 90.* Cambridge University Press, 1999, ISBN 0-521-57439-0.

Mit Fortran 90 wurden Vektor- und Matrix-Operationen standardisiert. Insbesondere für wissenschaftliche und numerische Berechnungen gibt es in FORTRAN umfangreiche Bibliotheken, die immer noch weit verbreitet sind, auch wenn eine zunehmende Menge an Routinen inzwischen nach C und C++ portiert wurde.

1.3. Compiler

Bis zum heutigen Zeitpunkt (Mai 2007) gibt es keinen Compiler der den aktuellen Fortran-Standard von 2003 vollständig unterstützt. Viele Compiler unterstützen jedoch schon Teile dieses Standards.

1.3.1. Kommerzielle Software

F95-Compiler gibt es für praktisch alle Computer, von Arbeitsplatzrechnern bis zu Supercomputern. Hersteller hierfür sind entweder die Computerhersteller wie z.B. IBM, Sun, HP, Intel oder aber spezialisierte Softwarehersteller wie z.B. Absoft, PGI, NAG, Lahey, Silverfrost/Salford. Reine F77-Compiler werden heute zumeist nicht mehr hergestellt, da Fortran 77 fast vollständig im Sprachstandard Fortran 95 enthalten ist.

Manche der oben genannten Compiler sind für Privatanwender bzw. nichtkommerzielle Nutzung kostenlos, zum Beispiel die Linux-Variante des Intel-Fortran-Compilers, Sun Studio Express oder für Microsoft-Windows der Compiler von Silverfrost/Salford.

1.3.2. Freie Software

Seit Version 4.0 der *GNU Compiler Collection* (GCC), die praktisch für alle Plattformen vorhanden ist, enthält diese ein Fortran 95-Frontend (gfortran). Außerdem existiert mit G95 ein weiterer freier Fortran 95-Compiler.

1.3.3. Übersetzer

Es gibt Programme, wie z.B. f2c, zur automatischen Übersetzung von Fortran in (allerdings kaum lesbares) C.

1.4. Referenzen

<references/>

Teil I.

Compiler

1.5. Was ist ein Compiler?

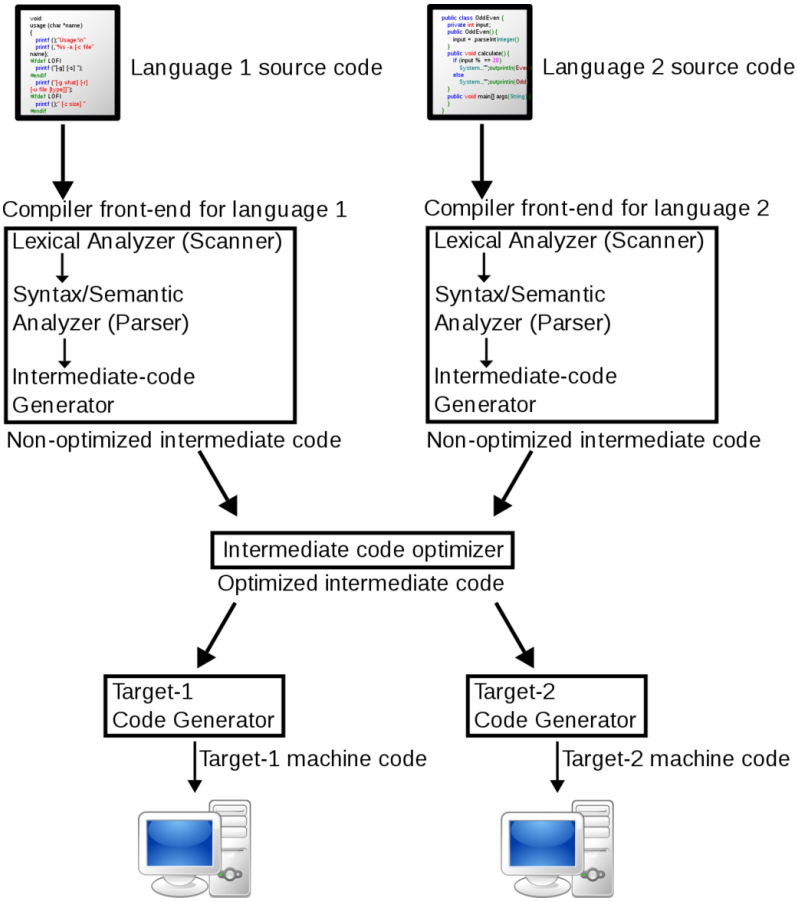


Abb. 3: Compilerschema

Ein Compiler (auch Kompilierer oder Übersetzer) ist ein Computerprogramm, das ein in einer Quellsprache geschriebenes Pro-

gramm - genannt Quellprogramm - in ein semantisch äquivalentes Programm einer Zielsprache (Zielprogramm) umwandelt. Üblicherweise handelt es sich dabei um die Übersetzung eines von einem Programmierer in einer Programmiersprache geschriebenen Quelltextes in Assemblersprache, Bytecode oder Maschinensprache. Das Übersetzen eines Quellprogramms in ein Zielprogramm durch einen Compiler wird als Kompilierung oder auch als Übersetzung bezeichnet.

Die Bezeichnung *Compiler* (engl. to compile: zusammenstellen) ist eigentlich irreführend. Ursprünglich bezeichnete das Wort Compiler Programme, die Unterprogramme zusammenfügen (etwa mit heutigen Linkern vergleichbar). Dies geht an der heutigen Kernaufgabe eines Compilers vorbei.

Verwandt mit einem Compiler ist ein Interpreter, der ein Programm nicht in die Zielsprache übersetzt, sondern Schritt für Schritt direkt ausführt.

1.6. Aufbau eines Compilers

Compiler werden in verschiedene Phasen gegliedert, die jeweils verschiedene Teilaufgaben des Compilers übernehmen. Sie werden sequentiell ausgeführt. Im Wesentlichen lassen sich zwei Phasen unterscheiden: das Frontend (auch *Analysephase*), das den Quelltext analysiert und daraus einen attributierten Syntaxbaum erzeugt, sowie das Backend (auch *Synthesephase*), das daraus das Zielprogramm erzeugt.

1.6.1. Frontend (auch „Analysephase“)

Im Frontend wird der Code analysiert, strukturiert und auf Fehler geprüft. Es ist auch selbst wieder in Phasen gegliedert:

Lexikalische Analyse

Die lexikalische Analyse zerteilt den eingelesenen Quelltext in zusammengehörende Token verschiedener Klassen, z. B. Schlüsselwörter, Bezeichner, Zahlen und Operatoren. Dieser Teil des Compilers heißt Scanner oder Lexer.

Ein Scanner benutzt gelegentlich einen separaten Screener, um Whitespace (also Leerzeichen, Tabulatorzeichen, Zeilenenden) und Kommentare zu überspringen.

Syntaktische Analyse

Die syntaktische Analyse überprüft, ob der eingelesene Quellcode ein korrektes Programm der zu übersetzenden Quellsprache ist, d. h. der Syntax (Grammatik) der Quellsprache entspricht. Dabei wird die Eingabe in einen Syntaxbaum umgewandelt. Dieser Teil wird auch als Parser bezeichnet.

Semantische Analyse

Die semantische Analyse überprüft die statische Semantik, also über die syntaktische Analyse hinausgehende Bedingungen an das Programm. Zum Beispiel muss eine Variable in der Regel deklariert worden sein, bevor sie verwendet wird, und Zuweisungen müssen mit kompatiblen (verträglichen) Datentypen erfolgen. Dies kann mit Hilfe von Attributgrammatiken realisiert werden. Dabei werden die Knoten des vom Parser generierten Syntaxbaums mit Attributen versehen, die Informationen enthalten. So kann zum Beispiel eine Liste aller deklarierten Variablen erstellt werden. Die Ausgabe der semantischen Analyse nennt man dann dekorierter oder attributierter Syntaxbaum.

1.6.2. Backend (auch „Synthesephase“)

Das Backend erzeugt aus dem vom Frontend erstellten attributierten Syntaxbaum den Programmcode der Zielsprache.

Zwischencodeerzeugung

Viele moderne Compiler erzeugen aus dem Syntaxbaum einen Zwischencode, der schon relativ maschinennah sein kann und führen auf diesem Zwischencode z. B. Programmoptimierungen durch. Das bietet sich besonders bei Compilern an, die mehrere Quellsprachen oder verschiedene Zielplattformen unterstützen. Hier kann der Zwischencode auch ein Austauschformat sein.

Programmoptimierung

Der Zwischencode ist Basis vieler Programmoptimierungen.

Codegenerierung

Bei der Codegenerierung wird der Programmcode der Zielsprache entweder direkt aus dem Syntaxbaum oder aus dem Zwischencode erzeugt. Falls die Zielsprache eine Maschinensprache ist, kann das Ergebnis direkt ein ausführbares Programm sein oder eine so genannte Objektdatei, die durch das Linken mit der Laufzeitbibliothek und evtl. weiteren Objektdateien zu einer Bibliothek oder einem ausführbaren Programm führt.

1.7. Geschichte

Die Geschichte des Compilerbaus wurde von den jeweils aktuellen Programmiersprachen und Hardwarearchitekturen geprägt. Der erste Compiler (A-0) wurde 1952 von der Mathematikerin Grace Hopper entwickelt. Weitere frühe Meilensteine sind 1954 der erste FORTRAN-Compiler und 1960 der erste COBOL-Compiler. Viele Architekturmerkmale heutiger Compiler wurden aber erst in den 1960er Jahren entwickelt.

2. Gfortran

2.1. Was ist gfortran?

gfortran (oder *GNU Fortran*) ist ein Fortran-Compiler-Frontend für die GNU Compiler Collection (GCC).

2.2. Installation

Vorcompilierte Pakete und Anleitungen zur Installation von *gfortran* finden sich gegliedert nach Betriebssystem und Prozessortyp auf: HTTP://GCC.GNU.ORG/WIKI/GFORTRANBINARIES[1]

2.3. Starten des gfortran-Compilers

2.3.1. MS Windows XP

Nach erfolgter Installation befinde sich das *gfortran*-Softwarepaket beispielsweise im Verzeichnis `C:\Programme\gfortran`. Zwecks Funktionstest wird die *Eingabeaufforderung* von MS Windows gestartet und der *gfortran*-Compiler aufgerufen:

1 HTTP://GCC.GNU.ORG/WIKI/GFORTRANBINARIES

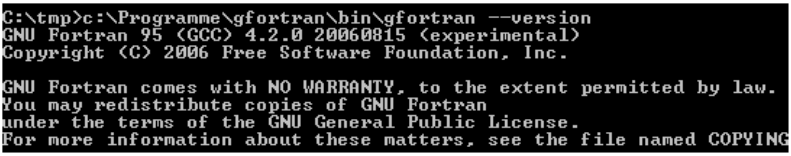


Abb. 4

Das Fortran-Programm für einen ersten konkreten Compilertest könnte so aussehen

```
program test
  write (*,*) 'Hallo Welt!'
end program test
```

Dieses Quellprogramm werde via Texteditor für dieses Beispiel unter dem Dateinamen `test.f90` im Verzeichnis `c:\tmp` gespeichert. Nun wird das Programm compiliert und gebunden

Abb. 5

und liegt dann unter dem Programmnamen `a.exe` als ausführbares Programm vor. Die Startanweisung für das Programm `a.exe` inklusive unmittelbar danach folgender Programmausgabe sieht so aus.

Abb. 6

Alternativ kann `a.exe` natürlich auch konventionell mittels *Windows-Explorer* gestartet werden. In diesem speziellen Fall wäre aber bis auf ein kurzes Aufblinken des Eingabeaufforderungs-Fensters nicht viel zu sehen, da das Programm nach der Textausgabe beendet wird.

2.3.2. Linux

Für die Linux-Variante des *gfortran*-Compilers gilt prinzipiell die selbe Vorgehensweise wie für die Windows-Variante.

Compilieren des Beispielprogrammes:

```
:/tmp> /disc1/programme/gfortran/bin/gfortran test.f90
```

Abb. 7

Starten des ausführbaren Programms `a.out` und Anzeige der Programmausgabe:

```
:/tmp> ./a.out
```

Abb. 8

```
 Hallo Welt!
```

Abb. 9

Durch entsprechende Nutzung des Linux-`PATH`-Mechanismus (z.B. symbolischer Link in ein PATH-Verzeichnis oder Aufnahme des `./gfortran/bin/`-Verzeichnisses in den `PATH`) kann die Angabe des gesamten Compilerpfades bei jedem `gfortran`-Compileraufruf entfallen.

```
:/tmp> gfortran test.f90
```

Abb. 10

2.4. Dateiendungen für Quelldateien

Mit *gfortran* lassen sich Programme verschiedener Fortran-Sprachstandardversionen kompilieren. Der Fortran-Typ wird üblicherweise durch die Dateiendung der Quelldatei festgelegt.

Dateiendung	Fortran-Version
.f	FORTRAN 77 (fixes Zeilenformat)
.f90	Fortran 90 (freies Zeilenformat)
.f95	Fortran 95 (freies Zeilenformat)
.f03	Fortran 2003 (freies Zeilenformat)
.F	FORTRAN 77 (fixes Zeilenformat) mit Preprocessing durch cpp
.F90	Fortran 90 (freies Zeilenformat) mit Preprocessing durch cpp
.F95	Fortran 95 (freies Zeilenformat) mit Preprocessing durch cpp
.F03	Fortran 2003 (freies Zeilenformat) mit Preprocessing durch cpp

2.5. Anwendung

In der Anwendung gleicht *gfortran* anderen GCC-Frontends (z. B. *gcc*, *g++* oder *g77*).

- Übersetzung einer Quelldatei in die ausführbare Datei *a.out* (bzw. *a.exe* auf MS Windows-Systemen):

```
gfortran bsp.f90
```

- Übersetzung einer Quelldatei in eine Objektdatei *bsp.o*:

```
gfortran -c bsp.f90
```

- Übersetzung einer Quelldatei in die ausführbare Datei *bsp*:

```
gfortran -o bsp bsp.f90
```

- Statisches Linken:

```
gfortran -static -o bsp bsp.f90
```

- Mehrere Quelldateien kompilieren und zu einer ausführbaren Datei linken:

```
gfortran -c bsp1.f90

gfortran -c bsp2.f90

gfortran -o bsp bsp1.o bsp2.o
```

- Mehrere Quelldateien in einer Anweisung kompilieren und zu einer ausführbaren Datei linken:

```
gfortran -o bsp bsp1.f90 bsp2.f90
```

2.6. Prüfung des Quellcodes auf Standardkonformität

Zur Sicherstellung einer strikten Standardkonformität kann die Option std mit folgenden Parametern verwendet werden:

Parameter	**Kommentar**
f95	Fortran 95
f2003	Fortran 2003 (noch nicht vollständig implementiert [Stand: Dez. 2005])

Parameter	Kommentar
gnu	Fortran mit GNU-Erweiterungen
legacy	

- Beispiele:

```
gfortran -std=f95 bsp.f

gfortran -std=f95 -W -Wall -pedantic bsp.f
```

2.7. gfortran-Weblinks

- GNU FORTRAN 95[2]
- GNU FORTRAN 95-WIKI[3]

2 HTTP://GCC.GNU.ORG/FORTRAN/
3 HTTP://GCC.GNU.ORG/WIKI/GFORTRAN

3. Fortran: G95

3.1. Über g95

Mit *g95* steht dem Fortran-Programmierer ein freier (im Sinne von Freier Software) Kommandozeilen-Compiler zur Verfügung, der für eine Vielzahl von Plattformen erhältlich ist.

3.2. Installation

3.2.1. Windows

Für die native Windows-Version existiert ein Self-Installer, g95-MinGW.exe, dessen Anweisungen man einfach folgen sollte. Da es sich um einen Kommandozeilen-Compiler handelt, sollte man entweder mit dem Windows-internen CMD.EXE umgehen können, oder man installiert sich eine komfortablere Kommandozeilen-Entwicklungsumgebung, z.B. MSYS[1] (dann aber am besten vor der g95-Installation) oder GNU UTILS FOR WIN32[2]. Alternativ erlaubt u.U. der zur Programmierung verwendete Editor bzw. die IDE eine komfortable Konfiguration, so dass g95 von dort aus aufgerufen werden kann.

1 HTTP://WWW.MINGW.ORG/MSYS.SHTML
2 HTTP://UNXUTILS.SOURCEFORGE.NET/

3.2.2. Unix-artige inkl. Cygwin und MacOSX

Für alle anderen Systeme steht ein komprimiertes tar-Archiv zur Verfügung. Dies kann an einem beliebigen Ort im Dateisystem entpackt werden, je nach Rechten z.B. in /usr/local oder im $HOME Verzeichnis. Für Linux z.B.:

```
tar -xzv -f g95-x86-linux.tgz -C /usr/local
```

oder

```
tar -xzv -f g95-x86-linux.tgz -C $HOME
```

Anschließend steht der Kompiler im Unterverzeichnis g95-install/bin/ mit einem eher unhandlichen Namen zur Verfügung. Daher sollte man sich einen symbolischen link anlegen, z.B. (je nachdem ob systemweit installiert werden soll oder nur für den Gebrauch als User):

```
ln -s /usr/local/g95-install/bin/*g95*
/usr/local/bin/g95
```

oder

```
ln -s $HOME/g95-install/bin/*g95*
$HOME/bin/g95
```

3.3. Bedienung

Die im Kapitel DAS COMPILER-FRONTEND *gfortran*[3] getroffenen Aussagen zu den Dateiendungen gelten auch für *g95*. Die Compil-

3 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20GFORTRAN%20

eraufrufoptionen sind ebenfalls in wesentlichen Teilen identisch zu jenen von *gfortran*.

So wird z.B. ein Programm, das nur aus einer Datei besteht, mit:

```
g95 -o bsp bsp.f
```

kompiliert.

3.4. Technischer Hintergrund

Wie GFORTRAN[4] baut *g95* auf der GNU Compiler Collection (GCC) auf. Streng genommen sind beide nur Frontends zu dem *GCC*-Backend. Allerdings präsentiert sich der installierte *g95* wie ein eigenständiger Compiler. Insbesondere bedarf es weder einer weiteren Installation einer bestimmten GCC-Version, noch treten Konflikte mit installierten GCC-Versionen auf. Dies spiegelt sich insbesondere in den flexiblen Installationsmöglichkeiten wieder.

g95 und *gfortran* basieren auf dem selben Programmcode. Im Jahr 2003 gabelten sich die Entwicklungszweige. *gfortran* ist nun Teil der *GCC*. Die Entwicklung von *g95* wird vom ursprünglichen Programmautor Andrew Vaught unabhängig davon weitergeführt.

3.5. Weblink

DAS G95-PROJEKT[5]

4 Kapitel 2 auf Seite 15

5 HTTP://WWW.G95.ORG

4. Fortran: Ifort

4.1. Allgemeines

Der *Intel Fortran Compiler for Linux* ist ein Fortran-Compiler, der für nicht-kommerzielle Zwecke auch in einer kostenfreien Variante verfügbar ist. Beeindruckend ist auch die diesem Softwarepaket beigegebene ausführliche Dokumentation, vor allem die knapp 900-seitige "Intel Fortran Language Reference". Im *Intel Fortran Compiler* sind neben den üblichen Fortran-Standardfunktionen auch eine Unmenge eigener Subroutinen und Funktionen implementiert.

4.2. Installation

1. Eine Downloadadresse und der Lizenzschlüssel für den *Intel Fortran Compilers for Linux* werden nach einer Registrierungsprozedur auf der Intel-Website per E-Mail übermittelt.
2. Download der Compiler-Software.
3. Entpacken der gepackten Datei (gunzip, tar).
4. Die eigentliche Installation des *Intel Fortran Compilers* erfolgt mittels install-Skript.

4.3. Dateiendungen für Quelldateien

Mit dem *Intel Fortran Compiler* lassen sich Programme verschiedener Fortran-Sprachstandardversionen kompilieren. Der Fortran-Typ wird üblicherweise durch die Dateiendung der Quelldatei festgelegt.

Dateiendung	**Fortran-Version**
.f, .for, .fnt, .i	FORTRAN 77 (fixes Zeilenformat)
.f90, .i90	Fortran 90/95 (freies Zeilenformat)
.F, .FOR, .FTN, .FPP, .fpp	FORTRAN 77 (fixes Zeilenformat) mit Preprocessing
.F90	Fortran 90/95 (freies Zeilenformat) mit Preprocessing

4.4. Anwendung

In der Anwendung gleicht der *Intel Fortran Compiler* dem *GNU Fortran Compiler*. Die offensichtlichsten Unterschiede sind:

- Die Intel Fortran Compiler-Software wird mittels *ifort* gestartet
- Die Intel Fortran Compiler-Software kennt die Dateiendungen *.f95* und *.F95* nicht
- Die Intel Fortran Compiler-Software enthält einen eigenen Kommandozeilendebugger *idb*.

```
idb bietet auch ein GUI .
Die Benutzung von anderen debugern wie gdb ist auch möglich.
```

- Übersetzung einer Quelldatei in die ausführbare Datei *a.out*:

ifort bsp.f90

Übersetzung einer Quelldatei in eine Objektdatei *bsp.o*:

ifort -c bsp.f90

- Übersetzung einer Quelldatei in die ausführbare Datei *bsp*:

  ifort -o bsp bsp.f90

- Mehrere Quelldateien kompilieren und zu einer ausführbaren Datei linken:

  ifort -c bsp1.f90

  ifort -c bsp2.f90

  ifort -o bsp bsp1.o bsp2.o

- Mehrere Quelldateien in einer Anweisung kompilieren und zu einer ausführbaren Datei linken:

  ifort -o bsp bsp1.f90 bsp2.f90

4.5. Weblink für den Intel Fortran Compiler for Linux

http://www.intel.com/cd/software/products/asmo-na/eng/compilers/flin/index.htm

Teil II.

Die Programmiersprache

4.6. Was ist eine Programmiersprache?

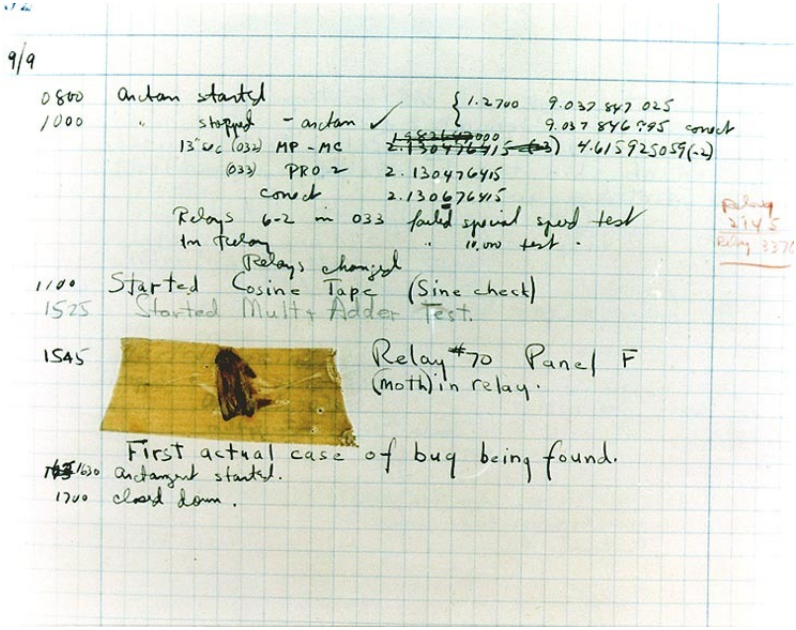


Abb. 11: Früher wurden Computer "debugged", heutzutage Computerprogramme

PROGRAMMIERSPRACHE[1]

In Kurzform: *Eine Programmiersprache ist eine Sprache zwecks Abfassung von Computerprogrammen.*

1 HTTP://DE.WIKIPEDIA.ORG/WIKI/PROGRAMMIERSPRACHE

4.7. Einordnung von Fortran

4.7.1. Generell

- Fortran ist eine **höhere Programmiersprache** (Higher Level Language, HLL, Programmiersprache der 3. Generation)
- Fortran ist eine **prozedurale Programmiersprache** (PROZEDURALE PROGRAMMIERUNG[2])
- Fortran ist eine **imperative Programmiersprache** (IMPERATIVE_PROGRAMMIERUNG[3])
- Fortran ist eine **objektorientierte Programmiersprache** (ab Fortran 2003, OBJEKTORIENTIERTE PROGRAMMIERUNG[4]).
- Fortran verwendet das Konzept der **starken Typisierung**. Fortran kennt **explizite und implizite Typisierung** (TYPISIERUNG (INFORMATIK)[5])
- Fortran ist eine sehr alte Programmiersprache, die aber laufend weiterentwickelt und somit den modernen Trends immer wieder angepasst wurde und wird.

4.7.2. Popularität

Die Popularität einer Programmiersprache einigermaßen fundiert zu bestimmen ist nicht einfach. Dennoch gibt es Institutionen, die das versuchen, sei es über die Anzahl von Einträgen in Suchmaschinen, Zugriffstatistiken für Internetseiten, Nutzerbefragun-

2 HTTP://DE.WIKIPEDIA.ORG/WIKI/PROZEDURALE%20PROGRAMMIERUNG

3 HTTP://DE.WIKIPEDIA.ORG/WIKI/IMPERATIVE_PROGRAMMIERUNG

4 HTTP://DE.WIKIPEDIA.ORG/WIKI/OBJEKTORIENTIERTE%20PROGRAMMIERUNG

5 HTTP://DE.WIKIPEDIA.ORG/WIKI/TYPISIERUNG%20%28INFORMATIK%29

gen oder auch zeitliche Veränderungen bei Buchverkäufen. Hier werden stellvertretend zwei dieser Statistiken angeführt:

Lt. TIOBE lag Fortran im September 2007 hinsichtlich Popularität (TPCI) an 20. Stelle von insgesamt 100 gelisteten Programmiersprachen. Das ist zwar weit hinter dem führenden Java oder dem zweitplatzierten C. Dennoch rangiert Fortran in dieser Statistik vor anderen bekannten Programmiersprachen, wie z.B. Pascal, Prolog, Haskell oder Smalltalk.

GULP liefert mit Stand 22.09.2007 eine Statistik "Wie viele IT-Freiberufler verfügen über fachliche Kenntnisse auf welchem Gebiet?". Für Fortran wurden 5860 Treffer gelistet. Führend sind auch hier Java, Basic und C mit über 20.000 Treffern.

Solche Statistiken sind natürlich mit Vorsicht zu geniessen, aber in Form eines groben Richtwerts "Daumen mal Pi" können sie schon einen ersten Eindruck von Verbreitung und Popularität einer Programmiersprache geben.

Im Bereich der numerische Datenverarbeitung, insbesondere auf Hochleistungsrechnern, ist Fortran gemeinsam mit C/C++ nach wie vor führend.

Siehe auch:

- TPCI - TIOBE PROGRAMMING COMMUNITY INDEX[6]
- GULP - STATISTIK - PROGRAMMIERSPRACHEN[7]

6 HTTP://WWW.TIOBE.COM/TPCI.HTM
7 HTTP://WWW.GULP.DE/STATISTIK/STATABPS.HTML

4.8. Ein Überblick über die historische Entwicklung von Fortran

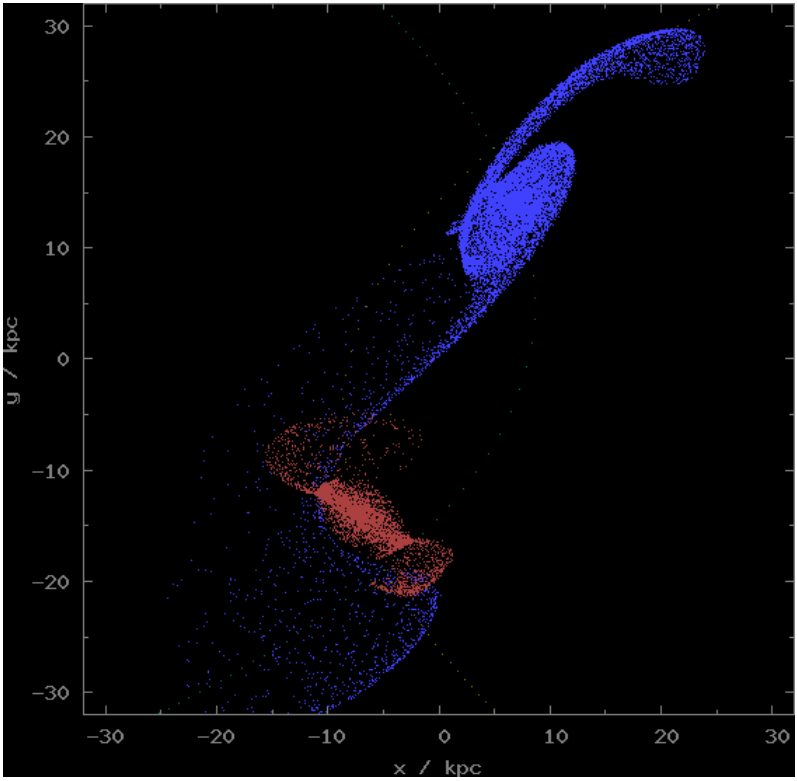


Abb. 12: Simulation von Galaxien mittels Fortran-Programm

Jahr	Version	Anmerkungen
1954 bis 1957	FORTRAN, FORTRAN I	entwickelt von einem IBM-Team unter Leitung von JOHN W. BACKUS[8], war dies die erste wirklich erfolgreiche höhere Programmiersprache
1957/58	FORTRAN II	Inline-Assembler, Kommentare, u.a.
1958	FORTRAN III	einige kleinere Änderungen, wurde aber nie offiziell freigegeben
1961/62	FORTRAN IV	eine verbesserte und erweiterte Version
1966	FORTRAN 66	die erste standardisierte Fortran-Version und gleichzeitig überhaupt die erste standardisierte höhere Programmiersprache
1978	FORTRAN 77	DO-Schleife, IF THEN-ELSE IF-ELSE-END IF, CHARACTER-Datentyp, u.a.

8 HTTP://DE.WIKIPEDIA.ORG/WIKI/JOHN%20WARNER%20BACKUS

Jahr	Version	Anmerkungen
1991	Fortran 90	free form style, Module, Zeiger, Datenverbund, u.v.m.
1997	Fortran 95	kleinere Änderungen
2003	Fortran 2003	OOP, C-Binding, u.a.
(2009)	(Fortran 2008)	geplant (Co-Arrays und kleinere Änderungen)

Siehe auch:

- THE HISTORICAL DEVELOPMENT OF FORTRAN[9]
- A BRIEF HISTORY OF FORTRAN/FORTRAN[10]
- FORTRAN I[11]
- HISTORY OF FORTRAN AND FORTRAN II[12]

9 HTTP://WWW.NSC.LIU.SE/~{}BOEIN/F77TO90/A7.HTML

10 HTTP://WWW.IBIBLIO.ORG/PUB/LANGUAGES/FORTRAN/CH1-{}1.HTML

11 HTTP://WWW.PAULGRAHAM.COM/HISTORY.HTML

12 HTTP://COMMUNITY.COMPUTERHISTORY.ORG/SCC/PROJECTS/FORTRAN/

Teil III.

FORTRAN 77

5. Programmaufbau

5.1. Beispiel: Hallo Welt

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM HALLO
C
C     Das typische Hallo Welt-Programm
C
      WRITE (*,*) 'Hallo Welt!'
      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

Wie Sie an diesem Beispiel ersehen, weist FORTRAN-77-Code eine charakteristische Struktur auf. Etwas seltsam mutet an, dass die Programmanweisungen nicht am Zeilenanfang stehen, sondern mehrere Leerzeichen eingerückt sind. Das ist obligatorisch, ansonsten würde sich das Programm nicht kompilieren lassen. Die Nummerierungen oberhalb und unterhalb der Programmanweisungen (12345678901...) gehören übrigens nicht zum FORTRAN-Code, sondern sollen nur die Spaltenstruktur von FORTRAN-77-Programmen verdeutlichen und die Orientierung etwas erleichtern.

Weiter ist zu erkennen, dass das Beispiel sehr kurz und aussagekräftig ausfällt. Die Anweisung in der ersten Zeile kennze-

ichnet die Programmeinheit als Hauptprogramm und gibt ihr die Bezeichnung `HALLO`. Die nächsten drei Zeilen sind Kommentarzeilen, erkennbar am Buchstaben in der ersten Spalte der Zeile. Dann folgt die Anweisung, einen String auf die Standardausgabe zu schreiben. Und schließlich signalisiert die `END`-Anweisung das Programmende.

5.2. Das Zeilenformat

Normalerweise gilt, dass jede FORTRAN-Anweisung in einer eigenen Zeile steht. Eine Spezialität von FORTRAN 77 ist die Spaltenorganisation eines Programmes: das fixe Zeilenformat. Es gibt fixe Spalten (Zeichenpositionen) in denen bestimmte Inhalte stehen müssen bzw. dürfen. Diese Art der Codeanordnung rührt von den Anforderungen der Lochkarte her. Bei den damaligen Großrechenanlagen war die der Eingabe von Programmen häufig nur in Form von Lochkartenstapeln möglich. FORTRAN 77 nimmt auf diese Beschränkung Rücksicht.

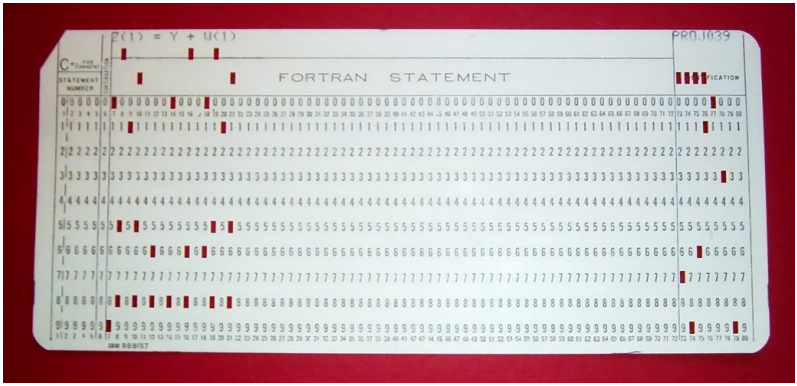


Abb. 13: Lochkarte mit Fortran-Statements

Der generelle Aufbau des fixen Zeilenformates von FORTRAN 77 ist wie folgt:

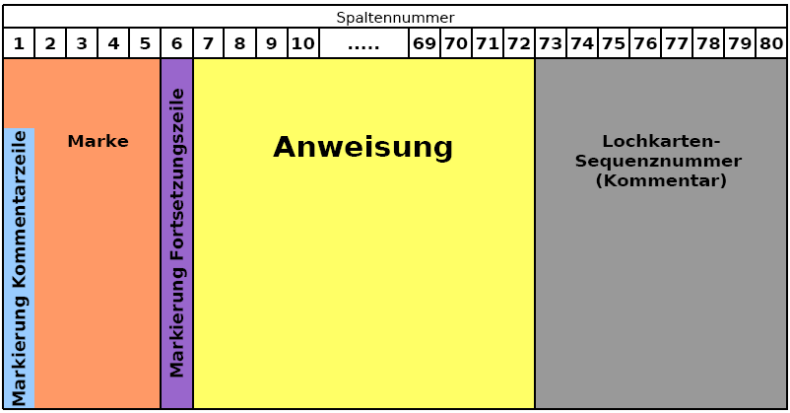


Abb. 14

In der nachstehenden Tabelle wird die Bedeutung der einzelnen Spalten detaillierter dargestellt.

Spalte	Inhalt	Bedeutung
1	C oder *	Kennzeichnet eine Kommentarzeile
1 bis 5	Eine Zahl 1 bis 99999	Anweisungsnummer (Marke)
6	Leerzeichen oder 0 (Null)	Beginn einer Anweisung (das ist der Normalfall)

Spalte	Inhalt	Bedeutung
6	sonstiges Zeichen	Fortsetzungszeile (standardmäßig sind bis zu 19 Fortsetzungszeilen erlaubt)
7 bis 72		FORTRAN-Befehl (Anweisung)
73 bis 80	beliebige Zeichen	Kommentar (ursprünglich für Lochkarten-Sequenznummern)

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP
* Leerzeilen werden übrigens wie Kommentarzeilen behandelt

* Eine 0 (Null) an der sechsten Position entspricht einem Leerzeichen,

* foerdert aber nicht gerade die Uebersichtlichkeit
     0A = 5
      B = 7
      C = A +
* Und jetzt kommt eine Fortsetzungszeile
     $B
      WRITE (*,*) C
      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Besondere Vorsicht ist bei langen Anweisungen geboten. Alles nach der 72 Zeichenposition wird nicht mehr als Teil der Anweisung aufgefasst. Im günstigsten Fall wirft der Compiler in einem solchen Fall bei der Übersetzung einen Syntaxfehler aus.

5.3. Die Programmstruktur für das Hauptprogramm

Ein Hauptprogramm weist immer folgende Struktur auf

1. **<code>PROGRAM** *prname*</code>
2. Vereinbarungsteil
3. Aktionsteil
4. `END`

prname ist ein symbolischer Name für das Hauptprogramm und kann mehr oder minder willkürlich festgelegt werden. Das erste Zeichen muss immer ein Buchstabe sein. Im Vereinbarungsteil werden z. B. die Variablen deklariert. Im Aktionsteil wird dann der eigentliche Programmablauf festgelegt. `END` kennzeichnet das Programmende. Theoretisch könnte im Hauptprogramm die erste Zeile (`PROGRAM prname`) auch komplett entfallen. In älteren Programmcodes wurde das durchaus auch so gehandhabt. Allerdings leidet darunter die Übersichtlichkeit des Programmes. Die `END`-Anweisung muss auf jeden Fall angegeben werden.

5.4. Der FORTRAN-Zeichenvorrat

FORTRAN-77-Programme bestehen standardmäßig nur aus folgenden Zeichen

- Großbuchstaben: A bis Z
- Ziffern: 0 bis 9
- 13 Sonderzeichen: + - * / = ( ) : , . ’ $ und dem Leerzeichen

Viele FORTRAN-77-Compiler akzeptieren auch Kleinbuchstaben. Zeichenkettenliterale können natürlich alle ASCII-Zeichen beinhalten.

5.5. Symbolische Namen

Standardmäßig dürfen symbolische Namen maximal sechs Zeichen lang sein. Als erstes Zeichen muss immer ein Buchstabe (A-Z) stehen, der Rest muss alphanumerisch sein (Buchstabe oder Ziffer). „Lustigerweise“ dürfen bei FORTRAN 77 Leerzeichen auch innerhalb eines symbolischen Namens auftreten.

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|   .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PR O GRA M BSP
*
* gfortran, g95, ifort etc. kompilieren diesen Code und
* bei der Programmausfuehrung wird auch das richtige
* Ergebnis angezeigt.
*
      ALPHA = 5
      BETA = 7
      GAMm a = A L PH   A + B E    TA
      WRI  TE (*,*) G A MM    A
      EN D


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|   .    8
```

Besonders unangenehm kann dieses Verhalten werden, wenn in einer Zählschleife anstelle eines Kommas irrtümlicherweise ein Punkt gesetzt wird, wenn also z. B. anstelle

```
DO 10 I = 1, 3
```

fälschlicherweise

```
DO 10 I = 1. 3
```

steht. Letzteres entspricht nämlich der Zuweisung der Zahl `1.3` an die Variable `DO10I`.

Gleiches gilt, wenn ähnlich wie in den Programmiersprachen C, C++ oder Java versucht wird, einer Variablen gleich in einer Deklarationszeile einen Wert zuzuweisen, z. B.

```
REAL A = 10.5
WRITE(*,*) A
```

Hier wird irgendein Wert ausgegeben, aber mit an Sicherheit grenzender Wahrscheinlichkeit nicht 10.500000, denn die Wertzuweisung erfolgte an die Variable `REALA` und nicht an `A`.

6. Datentypen, Variablen, Wertzuweisungen, Konstanten

Dieses Kapitel handelt von Datentypen, Variablen, Konstanten und der Wertzuweisung.

6.1. Datentypen

6.1.1. Arithmetische Datentypen

FORTRAN 77 unterscheidet standardmäßig zwischen vier arithmetischen Datentypen:

Datentyp	**Kommentar**	**Literale (Beispiele)**
`INTEGER`	Ganzzahlen	`15, -6500, 200 000 000`
`REAL`	Reelle Zahlen einfacher Genauigkeit	`3.1415, -5.5, .7E3, 12.5E-5`
`DOUBLE PRECISION`	Reelle Zahlen doppelter Genauigkeit	`3.1415D0, -5.5D0, .7D3, 12.5D-5`
`COMPLEX`	Komplexe Zahlen (zwei REAL-Zahlen)	`(3.1415, -5.5), (1.4, 7.1E4)`

6.1.2. Logischer Datentyp

Datentyp	Kommentar	Literale (alle)
`LOGICAL`	Logischer Datentyp (wahr oder falsch)	`.TRUE.`, `.FALSE.`

Manchmal sind in alten FORTRAN-Programmen auch folgende Schreibweisen zu finden, welche jedoch nicht standardkonform sind:

- `INTEGER*4`, `REAL*4`, `LOGICAL*4` (Standardgrößen)
- `INTEGER*1`, `LOGICAL*1`
- `INTEGER*2`, `LOGICAL*2`
- `REAL*8` (entspricht DOUBLE PRECISION)
- `COMPLEX*16` (komplexe Zahlen mit zwei DOUBLE-PRECISION-Elementen)

Die Zahlen geben den Speicherplatzbedarf in Byte an.

6.1.3. Zeichenketten

Datentyp	Kommentar	Beispiel (Konstante)
`CHARACTER*n`	Zeichenkette (String) mit einer Länge von *n* Zeichen	'Hallo, Welt!'
`CHARACTER`	Zeichenkette mit einer Länge von einem Zeichen (entspricht `CHARACTER*1`)	'A'

Beachte: Im Gegensatz zu vielen anderen Programmiersprachen werden Zeichenketten in FORTRAN 77 nicht in Anführungszeichen eingeschlossen, sondern in Apostrophe.

Tritt in einem String ein Apostroph auf, so muss dieses verdoppelt werden, z. B.

```
'Wie geht"s?'
```

Beispiel:

```
CHARACTER*5 STR
STR = 'Hallo'
```

Alternative Schreibweise:

```
CHARACTER STR*5
STR = 'Hallo'
```

6.2. Variablen

Eine Variable ist charakterisiert durch einen

- symbolischen Namen
- Datentyp
- Wert
- Speicherplatz

Beim Programmstart hat eine Variable keinen definierten Wert. Eine Variable kann ihren Datentyp auf zwei Arten erhalten, durch implizite oder explizite Typanweisung.

6.2.1. Implizite Typanweisung

Bei der impliziten Typanweisung bestimmt der Anfangsbuchstabe des Variablenbezeichners den Datentyp.

Anfangsbuchstabe der Variablen	**Impliziter Datentyp**
I, J, K, L, M oder N	INTEGER
alle restliche Buchstaben	REAL

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      B1 = 8.9
      C1 = 3.
      I1 = B1/C1

      WRITE (*,*) I1
C  Das Ergebnis ist 2, da I1 implizit als INTEGER definiert ist
      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Die Standardzuordnung der Anfangsbuchstaben kann durch das Schlüsselwort `IMPLICIT` auch geändert werden.

Es gibt noch ein weiteres Problem dieser impliziten Datentypzuordnung. Durch Tippfehler können unbeabsichtigt neue Variablen entstehen. Entschärft werden kann diese Tatsache durch folgende Festlegung im Vereinbarungsteil des Programms:

```
IMPLICIT LOGICAL (A-Z)
```

Dadurch werden alle Variablen mit implizit festgelegtem Datentyp automatisch zu Variablen mit logischem Datentyp. In vielen Fällen konnten so Tippfehler bei Variablennamen schnell eingegrenzt werden.

Beispiel:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP
      IMPLICIT LOGICAL (A-Z)
      REAL REE

      REE = 5.8
C  Tippfehler: REA anstatt REE
      WRITE (*,*) REA + 2.1

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Der Compilierungsversuch mit *g77*, *gfortran* und *g95* endet mit einer Fehlermeldung. Bei *gfortran* sieht das so aus

```
 In file test.f:9

      WRITE (*,*) REA + 2.1
                1
Error: Operands of binary numeric operator '+' at (1) are
LOGICAL(4)/REAL(4)
```

Doch der *Intel Fortran Compiler 9.0* zeigt die Grenzen der Sinnhaftigkeit der `IMPLICIT`-Anweisung in der dargestellten Art und Weise beim heutigen Einsatz von FORTRAN-77-Code auf.

Dieser Compiler akzeptiert den Beispielscode warnhinweislos und liefert bei Programmausführung den Wert `2.1`. Wirklich Abhilfe schafft also erst die `IMPLICIT NONE`-Anweisung. Diese legt eindeutig fest, dass ausschließlich die explizite Datentypfestlegung Verwendung finden soll. Allerdings ist `IMPLICIT NONE` erst ab Fortran-90/95-Standard. In einigen noch erhältlichen FORTRAN-77-Compilern, wie dem *g77*, ist diese Anweisung im Vorgriff auf den genannten neueren Standard bereits implementiert.

6.2.2. Explizite Typanweisung

Durch die Vorgabe von

datentyp variablenbezeichner

im Vereinbarungsteil des Programms wird der Datentyp einer Variablen explizit festgelegt. Die explizite Typanweisung hat gegenüber der impliziten Typanweisung Vorrang.

Beispiel:

```
0    .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      IMPLICIT LOGICAL (A-Z)
      REAL B
      REAL C
      REAL I

      B = 8.9
      C = 3.
      I = B/C

      WRITE (*,*) I
C  Das Ergebnis ist 2.966666
      END
```

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

6.3. Benannte Konstanten

Benannte Konstanten können durch das `PARAMETER`-Schlüsselwort festgelegt werden.

```
CHARACTER*5 STR
PARAMETER (PI=3.1415, PIFAK=PI/2., STR='Hallo')
```

Der zugewiesene Wert kann eine Konstante (Literal) oder eine schon definierte benannte Konstante sein. Der Datentyp muss vorher vereinbart werden oder ist implizit bekannt.

Für Zeichenketten ist im Zusammenwirken mit `PARAMETER` auch eine *-Schreibweise möglich. Dies erspart die explizite Angabe der Stringlänge.

Beispiel:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      CHARACTER*(*) A
      PARAMETER (A = 'Jetzt wird auch die Stringlaenge festgelegt')

      WRITE (*,*) A
C  Ausgabe: Jetzt wird auch die Stringlaenge festgelegt

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

6.4. Wertzuweisung

Wertzuweisungen haben wir schon kennengelernt:

```
variable = ausdruck
```

Beispiel:

```
K = 1
K = K + 2
```

Die Wertzuweisung an eine Variable ist, wie am vorigen Beispiel und auch nachfolgend zu ersehen, nicht zu verwechseln mit einer mathematischen Gleichung. Der Ausdruck

```
K + 2 = 5
```

wäre zwar mathematisch korrekt. Als Wertzuweisung in einem FORTRAN-Programm ist dies aber keine gültige Formulierung. `K + 2` ist kein zulässiger Ausdruck auf der linken Seite des Zuweisungsoperators (L-Wert).

Beachte: In FORTRAN 77 ist auch keine Kette von Wertzuweisungen möglich. Der folgende Ausdruck ist in FORTRAN 77 **nicht** erlaubt und liefert eine Fehlermeldung.

```
      I = J = K = 1.5
C Fehler!
```

7. Felder

Bei allem, was mehr oder weniger wie ein Vektor, eine Matrix oder eine sonstige Aneinanderreihung von gleichartigen Elementen aussieht, kann der Einsatz von Feldern (Arrays) sinnvoll sein.

7.1. Eindimensionale Felder

Für die Deklaration von eindimensionalen Feldern gibt es mehrere Möglichkeiten. Die Feldgrenzen müssen konstante Werte sein. Die Varianten werden nun anhand von Beispielen gezeigt.

7.1.1. Variante 1: Einfach

```
REAL ARR(10)
```

Beachte: Der Feldindex läuft hier von 1 bis 10 und nicht von 0 bis 9, wie es bei vielen modernen Hochsprachen der Fall ist.

7.1.2. Variante 2: Das `DIMENSION`-Schlüsselwort

```
REAL ARR
DIMENSION ARR(10)
```

7.1.3. Variante 3: Verwendung von benannten Konstanten

```
INTEGER MAXIND
PARAMETER (MAXIND=10)
REAL ARR(MAXIND)
```

Hier erfolgt die Festlegung der Feldgröße über eine benannte Konstante.

7.1.4. Variante 4: Explizite Angabe der Indexgrenzen

```
REAL ARR(0:9)
```

Hier wird Unter- und Obergrenze explizit angegeben. Der Index läuft nun von 0 bis 9. Auch negative Werte für die Indizes sind möglich, z. B.

```
REAL ARR(-4:5)
```

7.1.5. Beispiel

```
0    .|  1    .    2    .    3    .    4    .    5    .    6    .    7|   .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      INTEGER MAXIND
      PARAMETER (MAXIND=10)
      REAL ARR(MAXIND)
C ACHTUNG! Array startet mit dem Index 1
C ARR(0) waere ein Fehler!
      ARR(1) = 1.5
      ARR(2) = 2.5
      ARR(10) = 10.5

      WRITE (*,*) ARR(1)
```

```
C  1.5 wird ausgegeben

      WRITE (*,*) ARR(10)
C  10.5 wird ausgegeben
      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Ein

```
WRITE (*,*) ARR
```

listet den gesamten Feldinhalt.

```
 1.500000       2.500000       0.000000      3.9876625E-34   0.000000
    3.9902670E-34 -2.7682916E-05
-2.7269103E-05 -2.9040850E-05   10.50000
```

Im Beispielsfall wurden die Feldelemente `ARR(3)` bis `ARR(9)` nicht explizit vorbelegt. Sie sind deshalb undefinierten Inhalts und können bei jedem Programmaufruf andere Werte annehmen.

7.2. Mehrdimensionale Felder

Für mehrdimensionale Felder gelten die gleichen Varianten wie für eindimensionale Felder. Standardmäßig kann ein Feld bis zu sieben Dimensionen besitzen. Die Speicherreihenfolge ist spaltenorientiert. Das bedeutet, der erste Index variiert am schnellsten: $a_{11}, a_{21}, \ldots, a_{(n-1)m}, a_{nm}$

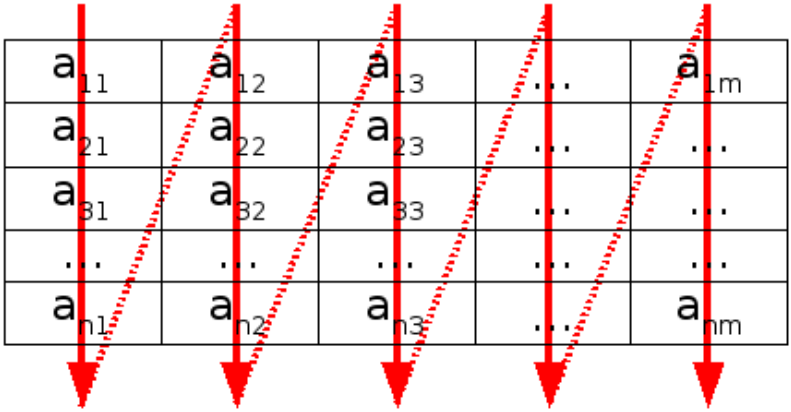


Abb. 15

7.2.1. Beispiel: Ein 2-dimensionales Feld

```
0    .|  1    .    2    .    3    .    4    .    5    .    6    .    7|   .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      CHARACTER*10 ARR(0:9, 2:5)

      ARR(0, 2) = 'Hallo'
      ARR(1, 2) = 'Welt'
C  ...
      ARR(9, 5) = 'Universum'
      WRITE (*,*) ARR(0, 2)
C  Hallo

      WRITE (*,*) ARR(9, 5)
C  Universum

      END
```

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

7.2.2. Beispiel: Spaltenorientierte Speicherreihenfolge

Die 3×3-Matrix $A = \begin{pmatrix} 1 & -5 & 0 \\ 40 & 3 & -2 \\ -1 & 9 & 65 \end{pmatrix}$ soll in ein Fortran-Programm eingelesen und wieder komplett ausgegeben werden. Zusätzlich soll auch der Wert des Feldelementes a_{23} (2. Zeile, 3.Spalte, Wert = -2) separat ausgegeben werden.

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      INTEGER ARR(3,3)
C Feldelemente einlesen
      WRITE (*,*) 'Werte (spaltenorientierte Eingabe):'
      READ (*,*) ARR

C Komplettes Feld ausgeben
      WRITE (*,*) 'Gesamtfeld = ' , ARR
C  a23 ausgeben
      WRITE (*,*) 'a23 = ', ARR(2,3)
      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Ein-/Ausgabe:

```
 Werte (spaltenorientierte Eingabe):
1
40
-1
```

```
-5
3
9
0
-2
65
 Gesamtfeld =            1          40          -1          -5
   3           9           0
     -2          65
 a23 =           -2
```

8. Arithmetische Ausdrücke

8.1. Arithmetische Operatoren

FORTRAN 77 kennt folgende arithmetische Operatoren

Operator	**Kommentar**	**Mathematische Entsprechung**
$A + B$	Addition	$A + B$
$A - B$	Subtraktion	$A - B$
$A * B$	Multiplikation	AB
A / B	Division	$\frac{A}{B}$
$A ** B$	Exponentiation	A^B

Mit dem Exponentiationsoperator (Potenzierung) war und ist FORTRAN 77 anderen Programmiersprachen einen Schritt voraus. Andererseits kennt FORTRAN 77 den aus vielen anderen Programmiersprachen bekannten Modulo-Operator nicht. Als Überkompensation gibt es für diesen Zweck die `MOD()`[1]-Funktion sowohl für Ganzzahlen, wie auch für Fließkommazahlen.

1 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%3A_Standardfunktionen%23Modulo

8.1.1. Operatorenpriorität

Die Priorität der arithmetischen Operatoren entspricht den mathematischen Gesetzmäßigkeiten.

- Klammerung vor allem anderen, z. B. `(A+B)*C` ⇔ `A*C+B*C`
- Exponentiation vor Punktrechnung, z. B. `A*B**C` ⇔ `A*(B**C)`
- Punktrechnung vor Strichrechnung, z. B. `A+B*C` ⇔ `A+(B*C)`

8.1.2. Berechnungsfolge bei gleicher Priorität

- Klammerung, Punktrechnung und Strichrechnung: →
  Beispiel: `A*B/C*D`⇔ `((A*B)/C)*D`
- Exponentiation: ←
  Beispiel: `A**B**C`⇔ `A**(B**C)`

Außerdem ist zu beachten, dass niemals zwei Operatoren direkt aufeinander folgen dürfen.

Beispiel: Der Ausdruck `1.5**-1` ist in FORTRAN 77 falsch und führt zu einer Fehlermeldung. Richtig ist `1.5**(-1)`

8.2. Ergebnisdatentyp

8.2.1. Operanden gleichen Datentyps

Bei Operanden gleichen Datentyps erhält das Ergebnis den Datentyp der Operanden.

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      REAL A
```

```
      A = 3/2
C  3 ist ein INTEGER und 2 ist auch ein INTEGER,
C  daher muss das Ergebnis auch ein INTEGER sein, also 1.
C  Die Zuweisung an die REAL-Variable A stellt das
C  Ergebnis nicht mehr richtig.

      WRITE (*,*) A
C  Ausgabe: 1.00000

      END
```

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

8.2.2. Implizite Typumwandlung bei Operanden gemischten Datentyps

Weisen die Operanden unterschiedliche Datentypen auf, so wird bei jeder Operation, falls nötig, das Ergebnis dem höherwertigen Datentyp angepasst.

INTEGER→ REAL→ DOUBLE PRECISION → COMPLEX

Beispiel:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      REAL A

      A = 3/2.
C  2. ist ein REAL. Jetzt stimmt das Ergebnis.

      WRITE (*,*) A
C  Ausgabe: 1.500000
```

```
      END


123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

8.2.3. Explizite Typumwandlung

FORTRAN 77 besitzt auch Funktionen zur expliziten Umwandlung des Datentyps. Diese werden im Kapitel STANDARDFUNKTIONEN[2] näher beschrieben.

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

       PROGRAM BSP

       REAL R
       COMPLEX C

       R = 2
       C = CMPLX(R)

       WRITE (*,*) C
 C Ausgabe: ( 2.000000 , 0.000000 )

       END


123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

2 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN_77%3A_STANDARDFUNKTIONEN%23DATENTYPUMWANDLUNG%20

9. Logische Ausdrücke

Logische Ausdrücke können zwei Zustände annehmen, *wahr* oder *falsch*. Diese werden in FORTRAN 77 durch die Literale `.TRUE.` oder `.FALSE.` dargestellt.

9.1. Logische Operatoren

Folgende Tabelle enthält alle in FORTRAN 77 bekannte logische Operatoren. Sie sind in der Reihenfolge ihrer Prioritäten absteigend geordnet.

Operator	**Kommentar**	**Mathematische Entsprechung**
`.NOT.` A	logisches NICHT	$\neg A$
A `.AND.` B	logisches UND	AB
A `.OR.` B	logisches ODER	$A \vee B$
A `.EQV.` B	logische Äquivalenz (XNOR)	$\overline{A \oplus B}$
A `.NEQV.` B	logische Antivalenz (XOR)	$A \oplus B$

9.2. Wahrheitstafel

A	B	.NOT. A	A .AND. B	A .OR. B	A .EQV. B	A .NEQV. B
.TRUE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.
.TRUE.	.FALSE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	.FALSE.	.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.

Beispiel:

```
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      LOGICAL L

      L = .TRUE.

      WRITE (*,*) .NOT. L
C  Ausgabe: F

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Beispiel:

```
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      LOGICAL A, B

      A = .TRUE.
      B = .FALSE.

      WRITE (*,*) A .NEQV. B
C  Ausgabe: T
```

```
      END

12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1   .   2   .   3   .   4   .   5   .   6   .   7|  .   8
```

10. Vergleichsausdrücke

Zum Vergleichen zweier arithmetischer Ausdrücke oder von Strings gibt es in FORTRAN 77 Vergleichsoperatoren. Das Ergebnis eines Vergleichs ist immer logischer Wert (`.TRUE.` oder `.FALSE.`).

10.1. Vergleichsoperatoren für arithmetische Typen

Operator	**Kommentar**	**Mathematische Entsprechung**
A `.LT.` *B*	less than (kleiner als)	$A < B$
A `.LE.` *B*	less equal (kleiner gleich)	$A \leq B$
A `.GT.` *B*	greater than (größer als)	$A > B$
A `.GE.` *B*	greater equal (größer gleich)	$A \geq B$
A `.EQ.` *B*	equal (gleich)	$A = B$
A `.NE.` *B*	not equal (ungleich)	$A \neq B$

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
```

```
      PROGRAM BSP

      INTEGER A, B

      A = 5
      B = 6

      WRITE (*,*) A .LT. B
C  Ausgabe: T

      END


1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

Beispiel:

```
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      CHARACTER*5 A, B

      A = 'Halli'
      B = 'Hallo'

      WRITE (*,*) A .LT. B
C  Ausgabe: T

      END


1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

Beim Rechnen mit Fließkommazahlen (Datentypen: `REAL`, `DOUBLE PRECISION`, `COMPLEX`) sind die systemimmanenten Rechenungenauigkeiten zu beachten. Aus diesem Grund sollten Fließkommazahlen nicht auf strikte (Un)Gleichheit geprüft werden, son-

dern Vergleiche sollten einen kleinen Toleranzbereich aufweisen: $x \pm \epsilon = y$.

Beispiel (hier mit $\epsilon = 0{,}00001$ und $y = 2$:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      X = LOG(A)
C  Statt...
      IF (X .EQ. 2)
C  besser
      IF (ABS(X - 2) .LT. .00001)


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

10.2. Zeichenkettenvergleiche

Das Ergebnis eines Vergleichs von Zeichenketten mittels Vergleichsoperatoren ist teilweise systemabhängig. Ausnahmen sind `.EQ.` und `.NE.`. Systemunabhängige Resultate sind durch Verwendung der entsprechenden LEXIKALISCHEN STANDARDFUNKTIONEN[1] erhältlich. Dort wird immer die Reihenfolge im ASCII-Zeichensatz verwendet.

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      IMPLICIT LOGICAL(A-Z)

      CHARACTER*15 A, B
      A = 'Hallö'
```

1 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN_77%3A_STANDARDFUNKTIONEN%23LEXIKALISCHE_FUNKTIONEN%20

```
      B = 'hallo'

      WRITE (*,*) 'A gleich B? ', A .EQ. B
      WRITE (*,*) 'A kleiner als B (Operator)? ', A .LT. B
      WRITE (*,*) 'A kleiner als B (Funktion)? ', LLT (A, B)
C  Ausgabe:
C   A gleich B?  F
C   A kleiner als B (Operator)?  T
C   A kleiner als B (Funktion)?  T
      END


123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

10.3. Operatorenpriorität

1. Klammerung
2. Arithmetische Operatoren
3. Vergleichsoperatoren
4. Logische Operatoren
5. Zuweisungsoperator

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      IMPLICIT LOGICAL(A-Z)

      REAL A, B, C
      LOGICAL X, RES
      A = 5.5
      B = -1.2
      C = 8.6
      X = .FALSE.
      RES = X .AND. A - B .GT. C .OR. A .LE. C
C entspricht infolge Op.priorität:
C   RES = ((X .AND. ((A-B) .GT. C)) .OR. (A .LE. C))
```

```
      WRITE (*,*) RES
C Ausgabe: T
      END


123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|   .    8
```

Vergleichsoperatoren und logische Operatoren finden in erster Linie bei VERZWEIGUNGEN UND SCHLEIFENBEDINGUNGEN[2] Verwendung.

2 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20FORTRAN%2077%3A%20VERZWEIGUNGEN%20UND%20SCHLEIFEN%20

11. Stringoperationen

FORTRAN 77 bietet vergleichsweise komfortable Operatoren zur Behandlung von Zeichenketten.

11.1. Verknüpfungsoperator

Operator	Kommentar
A // B	Operator zum Verknüpfen von Zeichenketten

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      CHARACTER*4 A, B*10

      A='How '
      B='do you do.'

      WRITE (*,*) A // B
C  Ausgabe: How do you do.

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

11.2. Teilketten

Ein String ist ein `CHARACTER`-Feld. Auch Teilketten einer solchen Zeichenkette können adressiert werden.

Prinzip	**Beschreibung**
stringname (*anfang*:*ende*)	von *anfang* bis *ende*
stringname (:*ende*)	vom ersten Zeichen bis *ende*
stringname (*anfang*:)	von *anfang* bis zum letzten Zeichen
stringname (*index*:*index*)	genau ein Zeichen an der Position *index*

Dabei muss *anfang* stets größer oder gleich Eins sein. *ende* darf nicht größer als die Länge der Zeichenkette sein. *index* muss sich stets zwischen Eins und der Länge der Zeichenkette befinden.

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      CHARACTER*10 A

      A='Hallo Welt'

      WRITE (*,*) A(2:4)
C  Ausgabe: all

      WRITE (*,*) A(5:)
C  Ausgabe: o Welt

      WRITE (*,*) A(:3)
C  Ausgabe: Hal

      END
```

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

Beispiel:

```
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      CHARACTER*10 A

      A='Hallo Welt'
      A(7:) = 'XYZ'

      WRITE (*,*) A
C  Ausgabe: Hallo XYZ

      END
```

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

Neben diesen Möglichkeiten sind in FORTRAN 77 auch einige Standardfunktionen für das Hantieren mit Zeichenketten vorgesehen. Diese sind im Kapitel STANDARDFUNKTIONEN[1] beschrieben.

1 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN_77%3A_STANDARDFUNKTIONEN%23STRINGFUNKTIONEN

12. Verzweigungen und Schleifen

12.1. GOTO

`GOTO` bewirkt einen Sprung zu einer bestimmten Anweisungsnummer.

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      GOTO 100
      WRITE (*,*) 'Hallo'
 100  WRITE (*,*) 'Welt'
C  Ausgabe: Welt

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

12.2. CONTINUE

`CONTINUE` ermöglicht bei Anweisungsnummern eine „leere“ Anweisung.

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      GOTO 100
      WRITE (*,*) 'Hallo'
 100  CONTINUE
C  keine Ausgabe

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

12.3. Bedingtes GOTO

Beim bedingten `GOTO` ist in Abhängigkeit von einer Integer-Variablen der Sprung zu einer bestimmten Anweisungsnummer möglich.

Beispiel: Eine „Switch“-Verzweigung

```
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      INTEGER I

      I=2
      GOTO (100, 200, 300), I
 100  WRITE (*,*) 'Hallo 1'
      GOTO 1000
 200  WRITE (*,*) 'Hallo 2'
      GOTO 1000
 300  WRITE (*,*) 'Hallo 3'
 1000 CONTINUE
C  Ausgabe: Hallo 2

      END
```

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

12.4. IF-Verzweigungen

12.4.1. Der IF-Einzeiler

IF (logischer ausdruck) anweisung

Beispiel:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      INTEGER I

      I=2
      IF (I .EQ. 2) WRITE (*,*) 'Hallo'
C  Ausgabe: Hallo

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

12.4.2. IF-THEN

IF (logischer ausdruck) THEN anweisungsblock END IF

Beispiel:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
```

```
      PROGRAM BSP

      INTEGER I

      I=2
      IF (I .EQ. 2) THEN
        WRITE (*,*) 'Hallo'
      END IF
C  Ausgabe: Hallo

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

12.4.3. IF-THEN-ELSE

IF (logischer ausdruck) THEN if-anweisungsblock ELSE else-anweisungsblock END IF

Beispiel:

```
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      INTEGER I

      I=333
      IF (I .GE. 444) THEN
        WRITE (*,*) 'Hallo'
      ELSE
        WRITE (*,*) 'Hola'
      END IF
C  Ausgabe: Hola

     END
```

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

12.4.4. ELSE-IF

IF (logischer ausdruck 1) THEN if-anweisungsblock 1 ELSE IF (logischerAusdruck 2) THEN if-anweisungsblock 2 ELSE IF (logischerAusdruck n) THEN if-anweisungsblock n ELSE else-anweisungsblock END IF

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      INTEGER I
      I = 2
      IF (I .EQ. 1) THEN
          WRITE (*,*) 'I ist eins'
      ELSE IF (I .EQ. 2) THEN
          WRITE (*,*) 'I ist zwei'
      ELSE
          WRITE (*,*) 'Ich weiß nicht was I ist'
      END IF
C Ausgabe: I ist zwei

      END
```

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

12.5. DO-Schleifen

Die `DO`-Schleife (Zählschleife) ist die einzige Schleife die FORTRAN 77 standardmäßig kennt.

DO nr zählvariable = startwert, endwert [, schrittweite] anweisungsblock nr CONTINUE

Beispiel:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      INTEGER I

      DO 100 I = 1, 10
        WRITE (*,*) I
 100  CONTINUE
C  Zeilenweise Ausgabe der Zahlen 1 bis 10
      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

12.6. Weitere Schleifen

Da FORTRAN 77 keine weiteren Schleifen kennt, müssen diese mit Hilfe einer IF-Verzweigung und einem `GOTO`-Befehl nachgebildet werden. Prominente Beispiele sind die While-Schleife und die Repeat-Until-Schleife.

12.6.1. While-Schleife

Beispiel:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      INTEGER I

      I=0

 10   IF(I .LT. 5) THEN
        WRITE (*,*) I
        I = I + 1
        GOTO 10
      END IF
C  Die Zahlen 0 bis 4 werden ausgegeben

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

12.6.2. Repeat-Until-Schleife

Beispiel:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      INTEGER I

      I=6

 10   CONTINUE
        WRITE (*,*) I
        I = I + 1
      IF (I .LT. 5) GOTO 10
C  Die Zahl 6 wird ausgegeben
```

```
      END


123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

12.7. Implizite Schleife

Bei Eingabe oder Ausgabe ist die Angabe einer impliziten Schleife möglich.

Beispiel:

```
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      INTEGER I

      WRITE (*,*) ('Hallo', I = 1, 10)
C  Ausgabe: HalloHalloHalloHalloHalloHalloHalloHalloHalloHallo

      END


123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

12.8. STOP

Die `STOP`-Anweisung beendet das Programm.

Beispiel:

```
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
```

```
      PROGRAM BSP

      WRITE (*,*) 'Vor Stop-Statement'
      STOP
      WRITE (*,*) 'Nach Stop-Statement'
C  Ausgabe: Vor Stop-Statement

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

13. Standardfunktionen

In diesem Kapitel werden für die Funktionsargumente und Rückgabewerte nachfolgende Kürzel verwendetet, die Aufschluss über die möglichen Datentypen geben.

Kürzel	Datentypen
i	`INTEGER`
r	`REAL`
d	`DOUBLE PRECISION`
x	`COMPLEX`
c	`CHARACTER*n`
l	`LOGICAL`

13.1. Datentypumwandlung

13.1.1. Umwandlung in INTEGER

- i = INT(i)

Abschneiden des Nachkommaanteils:

- i = INT(r)
- i = IFIX(r)
- i = IDINT(d)

- i = INT(x)

ASCII-Wert des Zeichens c:

- i = ICHAR(c)

13.1.2. Umwandlung in REAL

- r = REAL(i)
- r = FLOAT(i)
- r = REAL(r)
- r = SNGL(d)
- r = REAL(x)

13.1.3. Umwandlung in DOUBLE

- d = DBLE(i)
- d = DBLE(r)
- d = DBLE(d)
- d = DBLE(x)

13.1.4. Umwandlung in COMPLEX

- x = CMPLX(i)
- x = CMPLX(r)
- x = CMPLX(d)
- x = CMPLX(x)

13.1.5. Umwandlung in CHARACTER

Gibt das Zeichen zum ASCII-Wert i zurück.

- c = CHAR(i)

13.2. Mathematische Funktionen

13.2.1. Abschneiden des Nachkommaanteils

Ist das Argument größer Null, wird die nächstkleinere ganze Zahl zurückgegeben. Ist das Argument kleiner Null, wird die nächstgrößere ganze Zahl zurückgegeben.

- r = AINT(r)
- d = DINT(d)

13.2.2. Runden

Ist das Argument größer oder gleich Null, ist der Rückgabewert `INT(`*X*`+0.5)`. Ist das Argument kleiner Null, ist der Rückgabewert `INT(`*X*`-0.5)`

- r = ANINT(r)
- d = DNINT(d)
- i = NINT(r)
- i = IDNINT(d)

13.2.3. Absolutwert

- i = IABS(i)
- r = ABS(r)
- d = DABS(d)
- r = CABS(x)

13.2.4. Double Precision-Produkt

Rückgabewert ist *r1* × *r2* mit Datentyp `DOUBLE PRECISION`

- d = DPROD(r1, r2)

13.2.5. Modulo

Rückgabewert ist `zahl1 - INT(zahl1 /zahl2)*zahl2`

- i = MOD(i1, i2)
- r = AMOD(r1, r2)
- d = DMOD(d1, d2)

13.2.6. Vorzeichentransfer

Wenn die `zahl2 >= 0` ist, dann wird `|zahl1|` zurückgegeben. Wenn die `zahl2 < 0` ist, dann wird `-|zahl1|` zurückgegeben.

- i = ISIGN(i1, i2)
- r = SIGN(r1, r2)
- d = DSIGN(d1, d2)

13.2.7. Positive Differenz

Für `zahl1 > zahl2` ist der Rückgabewert `zahl1 - zahl2`. Für `zahl1 <= zahl2` wird Null zurückgegeben.

- i = IDIM(i1, i2)
- r = DIM(r1, r2)
- d = DDIM(d1, d2)

13.2.8. Maximum

Gibt den größten Argumentwert zurück.

- i = MAX0(i1, i2, ...)

- r = AMAX1(r1, r2, ...)
- d = DMAX1(d1, d2, ...)
- r = AMAX0(i1, i2, ...)
- i = MAX1(r1, r2, ...)

13.2.9. Minimum

Gibt den kleinsten Argumentwert zurück.

- i = MIN0(i1, i2, ...)
- r = AMIN1(r1, r2, ...)
- d = DMIN1(d1, d2, ...)
- r = AMIN0(i1, i2, ...)
- i = MIN1(r1, r2, ...)

13.2.10. Komplexe Zahlen

Gibt den Imaginärteil zurück:

- r = AIMAG(x)

Gibt die konjugiert komplexe Zahl zurück:

- x = CONJG(x)

13.2.11. Quadratwurzel

Gibt die Quadratwurzel zurück:

- r = SQRT(r)
- d = DSQRT(d)
- x = CSQRT(x)

13.2.12. Exponentialfunktion

Gibt natürliche Exponentialfunktion zurück:

- r = EXP(r)
- d = DEXP(d)
- x = CEXP(x)

13.2.13. Logarithmus naturalis

Gibt den natürlichen Logarithmus zurück:

- r = ALOG(r)
- d = DLOG(d)
- x = CLOG(x)

13.2.14. Dekadischer Logarithmus

Gibt den dekadischen Logarithmus zurück:

- r = ALOG10(r)
- d = DLOG10(d)

13.2.15. Winkelfunktionen

- r = SIN(r)
- d = DSIN(d)
- x = CSIN(x)

- r = COS(r)
- d = DCOS(d)
- x = CCOS(x)

- r = TAN(r)

- d = DTAN(d)

13.2.16. Arkusfunktionen

- r = ASIN(r)
- d = DASIN(d)

- r = ACOS(r)
- d = DACOS(d)

- r = ATAN(r)
- d = DATAN(d)

Gibt arctan($\frac{r1}{r2}$) zurück:

- r = ATAN2(r1, r2)
- d = DATAN2(d1, d2)

Diese Funktionen sind für ähnliche Werte der beiden Argumente erheblich genauer.

13.2.17. Hyperbelfunktionen

- r = SINH(r)
- d = DSINH(d)

- r = COSH(r)
- d = DCOSH(d)

- r = TANH(r)
- d = DTANH(d)

13.3. Zeichenketten-Funktionen

13.3.1. Länge

- i = LEN(c)

13.3.2. Index eines Teilstrings

Gibt die erste Position des Auftretens eines Teilstrings c2 in c1 zurück. c2 muss eine (benannte) Konstante sein.

- i = INDEX(c1, c2)

13.3.3. Lexikalische Funktionen

Hier wird unabhängig von der Plattform immer der ASCII-Zeichensatz als Grundlage verwendet.

Lexikalisch größer oder gleich (Rückgabewert ist `.TRUE.` wenn c1 >= c2):

- l = LGE(c1, c2)

Lexikalisch größer als (Rückgabewert ist `.TRUE.` wenn c1 > c2):

- l = LGT(c1, c2)

Lexikalisch kleiner oder gleich (Rückgabewert ist `.TRUE.` wenn c1 <= c2):

- l = LLE(c1, c2)

Lexikalisch kleiner als (Rückgabewert ist `.TRUE.` wenn c1 < c2):

- l = LLT(c1, c2)

Obige Funktionenauflistung basiert auf dem *Fortran 77 Sprachstandard X3J3/90.4, Kap.15: Functions and Subroutines* HTTP://WWW.FORTRAN.COM/F77_STD/RJCNF-15.HTMLSH-15.10[1]. Bei Unklarheiten sollte diese Originalqelle zu Rate gezogen werden, wenngleich die tabellarische Darstellung der "intrinsic functions" dort ziemlich gewöhnungsbedürftig ist.

1 HTTP://WWW.FORTRAN.COM/F77_STD/RJCNF-{}15.HTML#SH-{}15.10

14. Unterprogramme

Natürlich können in FORTRAN 77 auch eigene Unterprogramme erstellt werden.

14.1. Funktionsanweisung

Eine Funktionsanweisung (auch Anweisungsfunktion genannt) stellt die einfachste Möglichkeit dar, ein Unterprogramm in FORTRAN 77 zu realisieren. Eine Funktionsanweisung kann nur einen Ausdruck umfassen und gilt nur in der Programmeinheit in der sie definiert wurde.

Definieren einer Funktionsanweisung:

funktionsname([*formale parameter*]) = *ausdruck*

Aufruf der Funktion:

[*variable* **<code>=<code>**] *funktionsname*([*aktuelle parameter*])

Beispiel:

```
0   .| 1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      FUNK() = 5
```

```
      WRITE (*,*) FUNK ()
C  Ausgabe: 5.000000

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

Beispiel:

```
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      REAL A, B, C
      FUNK(A, B) = COS(A) * LOG(B)

      C = FUNK(3.1415, 2.)

      WRITE (*,*) C
C  Ausgabe: -0.6931472

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

14.2. FUNCTION

Soll eine Funktion mehrere Anweisungen umfassen, so genügt das Konzept der Funktionsanweisung nicht mehr. FORTRAN 77 kennt zu diesem Zweck das Schlüsselwort `FUNCTION`.

[*datentyp*] `FUNCTION` *funktionsname*`(`[*formale parameter*]`)`
anweisungen `END`

Aufgerufen wird eine derartige Funktion gleich wie eine Funktionsanweisung.

Beispiel:

Datei *bsp.f*:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

C  Funktionsaufruf
      WRITE(*,*) FUNK()
C  Ausgabe: 27.50000

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Datei *funk.f*:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      REAL FUNCTION FUNK()

      REAL TMP

      DO 10 I = 1,10
        TMP = TMP + I*0.5
 10   CONTINUE

      FUNK = TMP
      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Übersetzung mit *gfortran*:

```
gfortran bsp.f funk.f
```

Eine Funktion muss einen Wert zurückgeben. Welcher Wert das ist, wird durch eine Zuweisung an den Funktionsnamen erreicht. Wird am Anfang des Funktionskopfes kein Datentyp explizit vorgegeben, so gelten die Regeln für die implizite Datentypvergabe.

Mit Hilfe des Schlüsselwortes `RETURN` kann eine Funktion auch vor dem Funktionsende verlassen werden.

Beispiel:

Datei *bsp.f*:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

C  Funktionsaufruf
      WRITE(*,*) FUNK(3)
C  Ausgabe: 1.500000

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Datei *funk.f*:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      FUNCTION FUNK(I1)

      IF(I1 .LE. 5) THEN
        FUNK = 1.5
        RETURN
      END IF

      FUNK = SIN(I1*0.5)
```

```
      END

12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

14.3. SUBROUTINE

Eine Subroutine besitzt im Gegensatz zu einer Funktion keinen Datentyp und gibt keinen Wert zurück.

`SUBROUTINE` *subroutinenname*`(`[*formale parameter*]`)` *anweisungen* `END`

Aufruf der Subroutine:

`CALL` *subroutinenname*`(`[*aktuelle parameter*]`)`

Beispiel:

Datei *test.f*

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP
      CALL SUB
      END

12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

Datei *sub.f*

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
```

```
      SUBROUTINE SUB
      WRITE (*,*) 'Hallo Welt!'
      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0    .|  1    .    2    .    3    .    4    .    5    .    6    .    7|   .    8
```

Übersetzung mittels *gfortran*:

```
gfortran -c sub.f
gfortran -c test.f
gfortran  test.o sub.o
```

Anzeige auf der Standardausgabe:

```
Hallo Welt!
```

Auch eine Subroutine kann mittels `RETURN` vorzeitig verlassen werden.

Die aktuellen und formalen Parameter müssen hinsichtlich Datentyp, Anzahl, Reihenfolge übereinstimmen. Alle Namen und Variablen in einer Programmeinheit (Subroutine, Funktion oder Hauptprogramm) sind grundsätzlich nur lokal in der jeweiligen Programmeinheit bekannt. Über die Unterprogrammparameter können aber sehr wohl Werte in der aufrufenden Programmeinheit geändert werden.

Beispiel:

Datei *bsp.f*:

```
0    .|  1    .    2    .    3    .    4    .    5    .    6    .    7|   .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP
```

```
      REAL A = 2.0

      CALL SUB(A)

      WRITE(*,*) A
C  Ausgabe: 10

      END


123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

Datei *sub.f*:

```
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890

      SUBROUTINE SUB(X)

      REAL X
      REAL A

C  Unterprogrammparameter
      X = 10

C lokale Variable
      A = 500

      END


123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

14.4. Felder als Parameter

Beispiel: Übergabe eines ganzen Feldes

Datei *bsp.f*:

```
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      INTEGER FELD(3,3)
      INTEGER CNT

      CNT = 1

      DO 10 I = 1, 3
        DO 20 J = 1, 3
          FELD(J,I) = CNT
          CNT = 1 + CNT
 20     CONTINUE
 10   CONTINUE

C  Unterprogrammaufruf
      CALL SUB(FELD)
C  Ausgabe: 1   2   3   4   5   6   7   8   9

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Datei *sub.f*

```
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      SUBROUTINE SUB(ARR)

      INTEGER ARR(3, 3)

      WRITE(*,*) ARR

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Beispiel: Übergabe einer Feld-Teilkette

Datei *bsp.f*:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      INTEGER FELD(3,3)
      INTEGER CNT

      CNT = 1

      DO 10 I = 1, 3
        DO 20 J = 1, 3
          FELD(J,I) = CNT
          CNT = 1 + CNT
 20     CONTINUE
 10   CONTINUE

C  Unterprogrammaufruf
      CALL SUB(FELD(1:2,2:3))
C  Ausgabe: 4           5           7           8

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Datei *sub.f*

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      SUBROUTINE SUB(ARR)

      INTEGER ARR(0:1, 0:1)

      WRITE(*,*) ARR

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Beispiel: Übergabe eines Feld-Einzelelements

Datei *bsp.f*:

```
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP
      INTEGER FELD(3,3)
      INTEGER CNT

      CNT = 1

      DO 10 I = 1, 3
        DO 20 J = 1, 3
          FELD(J,I) = CNT
          CNT = 1 + CNT
 20     CONTINUE
 10   CONTINUE

C  Unterprogrammaufruf
      CALL SUB(FELD(1,2))
C  Ausgabe: 4

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

Datei *sub.f*

```
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      SUBROUTINE SUB(ARR)

      INTEGER ARR

      WRITE(*,*) ARR

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

14.5. Prozeduren als Parameter

Auch Prozeduren können als Parameter übergeben werden.

Standardfunktionen werden dazu folgendermaßen im Vereinbarungsteil gekennzeichnet:

Aufruf der Subroutine:

INTRINSIC *namensliste*

Eigene Funktionen oder Subroutinen mit:

EXTERNAL *namensliste*

Beispiel:

Datei *bsp.f*:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      REAL PI
      PARAMETER(PI=3.1415927)

C  intrinsic functions
      INTRINSIC SIN, COS

C  Unterprogrammaufrufe
      CALL SUB(SIN, PI)
C  Ausgabe: 0.000000
      CALL SUB(COS, PI)
C  Ausgabe: -1.000000

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Datei *sub.f*:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|   .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      SUBROUTINE SUB(FUNK, X)

      REAL FUNK, X

      WRITE(*,*) NINT(FUNK(X)*1000)/1000.0

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|   .    8
```

14.6. COMMON

Mit `COMMON` läßt sich ein gemeinsamer Datenbereich für mehrere Programmeinheiten realisieren.

Unbenannter COMMON:

`COMMON` *variablenliste*

`COMMON /`*name*`/` *variablenliste*

Beispiel:

Datei *bsp.f*:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|   .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

     PROGRAM BSP

      REAL A, B, C, D
      COMMON  A, B, C
      COMMON /C1/ D
```

```
      A = 4.0
      B = 5.0
      C = 6.0

      CALL SUB

      WRITE (*,*) A, B, C, D
C  Ausgabe: 3.330000   4.440000   6.000000   5.550000

      END
```

```
1234567890123456789012345678901234567890123456789012345678901234567890123456789
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Datei *sub.f*:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
1234567890123456789012345678901234567890123456789012345678901234567890123456789

      SUBROUTINE SUB

      REAL X, Y, Z
      COMMON X, Y
      COMMON /C1/ Z

      X = 3.33
      Y = 4.44
      Z = 5.55

      END
```

```
1234567890123456789012345678901234567890123456789012345678901234567890123456789
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

14.7. ENTRY

Mittels `ENTRY` kann gezielt in ein Unterprogamm gesprungen werden. Dieses Konstrukt widerspricht aber einer strukturierten Programmierung und sollte nicht verwendet werden.

`ENTRY` *entryname*[`(`[*formale parameter*]`)`]

Der Aufruf entspricht dem einer Subroutine:

`CALL` *entryname*[`(`[*aktuelle parameter*]`)`]

Beispiel:

Datei *bsp.f*

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|   .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      CALL SUB
C  Ausgabe: Hallo
C           Welt!

      CALL E1
C  Ausgabe: Welt!

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|   .    8
```

Datei *sub.f*:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|   .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      SUBROUTINE SUB

      WRITE(*,*) 'Hallo'
```

```
      ENTRY E1
      WRITE (*,*) 'Welt!'

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

14.8. SAVE

Durch ein SAVE-Statement in Unterprogrammen behalten die lokalen Variablen ihren jeweiligen Wert auch nach Verlassen des Unterprogrammes. Dieses Konstrukt ist meist nicht notwendig, da die meisten FORTRAN-Compiler dieses Verhalten ohnehin automatisch aufweisen (siehe auch Kapitel DATA[1] zwecks Initialisierung von Variablen).

SAVE [*variablenliste*]

Beispiel:

Datei *bsp.f*:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      CALL SUB
C  Ausgabe: 1.000000

      CALL SUB
C  Ausgabe: 2.000000
```

1 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%3A%20Unterprogramme%23DATA

```
      CALL SUB
C  Ausgabe: 3.000000

      END


123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

Datei *sub.f*:

```
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890

      SUBROUTINE SUB

      REAL A
      SAVE

      A = A + 1

      WRITE(*,*) A

      END


123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

14.9. DATA

`DATA` dient zur Wertinitialisierung von Variablen vor der Programmeinheitausführung. Diese Anweisung ist also nicht gleichzusetzen mit einer Wertzuweisung.

Beispiel:

`DATA` [*variablenliste*] / *variablenwerte*/

Beispiel:

Datei *bsp.f*:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      CALL SUB
C  Ausgabe: 1.000000

      CALL SUB
C  Ausgabe: 2.000000

      CALL SUB
C  Ausgabe: 3.000000

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

Datei *sub.f*:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      SUBROUTINE SUB

      REAL A
      DATA A /0.0/

      A = A + 1

      WRITE(*,*) A

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

Unterschied zwischen Wertinitialisierung und Wertzuweisung:

	Wertinitialisierung	**Wertzuweisung**
Code	PROGRAM BSP CALL SUB CALL SUB CALL SUB END SUBROUTINE SUB REAL A DATA A /3.0/ A = A + 1 WRITE(*,*) A END	PROGRAM BSP CALL SUB CALL SUB CALL SUB END SUBROUTINE SUB REAL A A = 3.0 A = A + 1 WRITE(*,*) A END
Ausgabe	4.000000 5.000000 6.000000	4.000000 4.000000 4.000000

15. Ein- und Ausgabe

15.1. READ

Abb. 16

Die `READ`-Anweisung dient dem Einlesen von Daten. Typisches Beispiel ist die Dateneingabe mittels Tastatur. Formal sieht eine `READ`-Anweisung so aus:

```
READ([UNIT=]unit, [FMT=]fmt [, ERR=err] [, END=end] [, IO-
STAT=iostat]) [eingabeliste]
```

- unit ... Nummer der Eingabeeinheit (ist systemabhängig), Sternoperator oder auch die einer Datei mittels `OPEN`-Anweisung zugeordnete Nummer.
- fmt ... Anweisungsnummer zu einer `FORMAT`-Anweisung oder Sternoperator
- err ... Tritt während der Eingabe ein Fehler auf, so wird zu dieser Anweisungsnummer gesprungen

- end ... Nach dem Einlesen des letzten Datensatzes wird zu diese Anweisungsnummer gesprungen
- iostat ... READ-Status

Listengesteuerte Eingabe auf der Standardeingabe (normalerweise die Tastatur):

```
READ (*,*) A, B, C
```

Alternativ kann das auch so geschrieben werden:

```
READ (UNIT=*, FMT=*) A, B, C
```

Beim *Intel Fortran Compiler*, *gfortran* und *g95* ist auch `UNIT = 5` als `stdin` (Tastatur) vorbelegt. Das Einlesen aus Dateien und die Einstellung des Formates werden später erläutert.

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP
      INTEGER I(5)
C  Einlesen in ein Feld (UNIT ... Standardeingabe, FMT ...
 listengesteuert)
      READ (*,*) I
C     ...

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

Kurze Erläuterung zu IOSTAT:

Wert	Erläuterung

Wert	Erläuterung
0	kein Fehler
positiver Wert (systemabhängig)	Fehler
negativer Wert (systemabhängig)	*End Of File* und kein Fehler

Beispiel:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP
      INTEGER I
      INTEGER ST
C  Einlesen eines Wertes
      READ (*, *, IOSTAT=ST) I

C  Ausgabe des IO-Status
      WRITE (*,*) 'IO-Status:', ST

      END



12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Ausgabe:

Für Eingabe: 5 → 0
Für Eingabe: 5.3 → Positiver Wert = Fehler

15.2. WRITE

Abb. 17

Die `WRITE`-Anweisung dient der Datenausgabe. Typisches Beispiel ist die Anzeige von Daten auf dem Bildschirm. Formal sieht eine `WRITE`-Anweisung so aus:

```
WRITE([UNIT=]unit, [FMT=]fmt [, ERR=err] [, IOSTAT=iostat])
[ausgabeliste]
```

- unit ... Nummer der Ausgabeeinheit (ist systemabhängig), Sternoperator oder auch die einer Datei mittels `OPEN`-Anweisung zugeordnete Nummer.
- fmt ... Anweisungsnummer zu einer `FORMAT`-Anweisung oder Sternoperator
- err ... Tritt während der Ausgabe ein Fehler auf, so wird zu dieser Anweisungsnummer gesprungen
- iostat ... WRITE-Status

Listengesteuerte Ausgabe auf der Standardausgabe (normalerweise der Bildschirm):

```
WRITE (*,*) A, B, C
```

Alternativ kann das auch so geschrieben werden:

```
WRITE (UNIT=*, FMT=*) A, B, C
```

Beim *Intel Fortran Compiler*, *gfortran* und *g95* sind auch

- `unit=0` als `stderr` (Bildschirm) und
- `unit=6` als `stdout` (Bildschirm)

vorbelegt. Bezüglich `IOSTAT` gilt auch hier der im vorigen Abschnitt kurz geschilderte Sachverhalt. Die Ausgabe in Dateien und die Einstellung des Formates werden nachfolgend erläutert.

Beispiel:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP
      INTEGER I(5)
C     I(1) = ...
C     ...
C Ausgabe der Feldwerte (UNIT ... Standardausgabe, FMT ...
 listengesteuert)
      WRITE (*,*) I
C     ...

      END



12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

15.3. Formatierung

Abb. 18

Die Ein- und Ausgabeformatierung kann beeinflusst werden. Zu diesem Zweck gibt es die `FORMAT`-Anweisung.

... (..., FMT = *marke*, ...) ... *marke* FORMAT (*formatliste*)

Alternativ dazu kann auch direkt bei der `FMT`-Option die Formatliste bekanntgemacht werden.

... (..., FMT = '(formatliste)', ...) ...

15.3.1. Formatlistenelemente

Formatspezifizierer	**Kommentar**
Ix[.z]	Ganzzahl mit einer Feldlänge von x Zeichen. z gibt die Mindestanzahl der auszugebenden Zeichen an (Feld wird, wenn nötig, mit führenden Nullen aufgefüllt).

Formatspezifizierer	Kommentar
Fx.y	Fixkommazahl mit einer Gesamtfeldlänge von x Zeichen. y ist die Anzahl der Nachkommastellen (Vorzeichen und Dezimalpunkt müssen in der Gesamtfeldlänge berücksichtigt werden).
Ex.y	Gleitkommazahl mit einer Gesamtfeldlänge von x Zeichen. y ist die Anzahl der Nachkommastellen. (Vorzeichen, Dezimalpunkt und die Zeichen für den Exponenten müssen in der Gesamtfeldlänge berücksichtigt werden).
Dx.y	-"-
A	Ein Zeichenkette.
Ax	Eine Zeichenkette mit x Zeichen.
Lx	Ein logischer Wert, T bzw. F
xX	x Leerzeichen.

Obige Tabelle der Formatlistenelemente ist nicht vollständig. Fortran kennt noch weitere Formatierungsmöglichkeiten. Die Ausgabe erfolgt normalerweise rechtsbündig. Reicht die Gesamtfeldlänge bei numerischen Werten nicht aus, so werden anstelle einer Zahl Sternchen angezeigt.

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
```

```
      PROGRAM BSP

      INTEGER A

      A = 999
      WRITE (*, 3333) A
C  Ausgabe: 999

      A = -999
      WRITE (*, 3333) A
C  Ausgabe: ***

 3333 FORMAT (I3)

      END
```

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|   .    8
```

Weitere Formatierungsbeispiele:

Code	**Ausgabe**
WRITE(*, 999) 1234 WRITE(*, 999) 1234567 WRITE(*, 999) 1234567890 999 FORMAT(I9.6)	001234 1234567 *********
WRITE(*, 999) 555.6666 WRITE(*, 999) +5.6 WRITE(*, 999) -55.666E7 WRITE(*, 999) -55555.666 999 FORMAT(F9.3)	555.667 5.600 ********* *********

Code	Ausgabe
WRITE(*, 999) 555.6666 WRITE(*, 999) +5.6 WRITE(*, 999) -55.666E7 WRITE(*, 999) -55555.666 999 FORMAT(E9.3)	0.556E+03 0.560E+01 -.557E+09 -.556E+05
WRITE(*, 999) 'Hallo' WRITE(*, 999) 'ABCDEFGHIJKL' WRITE(*, 888) 'ABCDEFGHIJKL' 888 FORMAT(A) 999 FORMAT(A10)	Hallo ABCDEFGHIJ ABCDEFGHIJKL
WRITE(*, *) 'FORTRAN', '77' WRITE(*, 999) 'FORTRAN', '77' 999 FORMAT(A, 1X, A)	FORTRAN77 FORTRAN 77
WRITE(*, 888) 'FORTRAN', '77' WRITE(*, 999) 'FORTRAN', '77' 888 FORMAT(A, T3, A) 999 FORMAT(A, T20, A)	FO77RAN FORTRAN 77
WRITE(*, 999) 'FORTRAN', '77' 999 FORMAT(A, /, A)	FORTRAN 77
WRITE(*, 999) 34.56 WRITE(*, *) 34.56 C SP ... Sign Plus (+) 999 FORMAT(SP, F12.3)	+34.560 34.56

15.3.2. Wiederholung von Formatteilen

Beispiel:

```
      WRITE (*, 100) 'abc', 10.3, 'xxx', 23.4
100   FORMAT (2(A3, F6.1))
```

15.3.3. WRITE etwas anders

Beispiel:

```
      WRITE (*, 100)
100   FORMAT ('Hallo', 1X, 'Welt!')
```

15.4. Dateien

Abb. 19

15.4.1. Datensatz

Datensätze können in folgender Form auftreten:

- Formatierter Datensatz: Textdatensatz
- Unformatierter Datensatz: Datensatz in einer maschineninternen Form.
- Dateiendesatz

15.4.2. Datei

Für FORTRAN 77 ist alles eine Datei, das durch READ oder WRITE bearbeitbar ist.

Zugriffsmethoden:

- Sequentieller Zugriff: Lesen ab Beginn der Datei (file) und dann immer den nächsten Datensatz einlesen. Geschrieben wird jeweils ans Dateiende. Auf interne Dateien kann nur sequentiell zugegriffen werden.
- Direkter Zugriff: Bearbeiten in beliebiger Reihenfolge durch Angabe der Satznummer.

Dateitypen:

- Externe Datei: Eine konventionelle Datei
- Interne Datei: CHARACTER-Variable oder -Feld.

Dateien haben im Betriebssystem einen Dateinamen. In FORTRAN wird eine Datei über eine Dateinummer (unit) angesprochen. Die Zuordnung erfolgt mit dem Befehl `OPEN`.

15.4.3. OPEN

Zum Öffnen einer externen Datei dient die `OPEN` -Anweisung.

OPEN (liste)

mit folgender *liste*

Element	**Kommentar**
[UNIT =] x	x ist eine Dateinummer (0 bis 99)
FILE = x	x ist der externe Dateiname
IOSTAT = x	x ist 0 wenn OPEN fehlerfrei ausgeführt wurde, ansonsten eine systemabhängige Fehlernummer
ERR = x	Bei Fehler Sprung zur Anweisungsnummer x
STATUS = x	Dateistatus: 'OLD' ... Datei existiert bereits 'NEW' ... Datei wird neu erzeugt 'SCRATCH' ... namenlose temporäre Datei 'UNKNOWN' ... System bestimmt Dateistatus selbst
ACCESS = x	Zugriffsmethode: 'SEQUENTIAL' ... Sequentielle Datei 'DIRECT' ... direkter Zugriff
FORM = x	Format: 'FORMATTED' oder 'UNFORMATTED'
RECL = x	Datensatzlänge (positive Zahl, ACCESS='DIRECT', in Bytes bzw. bei formatierten Dateien in Characters)

Element	Kommentar
BLANK = x	'NULL' (ignorieren von Leerzeichen bei numerischen Werten) oder 'ZERO' (Leerzeichen bei numerischen Werten als 0 interpretieren)

Eingestellte Vorgabewerte sind:

- STATUS = 'UNKNOWN'
- ACCESS = 'SEQUENTIAL'
- FORM = 'FORMATTED'
- BLANK = 'NULL'

Wird `ACCESS='DIRECT'` gesetzt, so gilt `FORM='UNFORMATTED'` als Vorgabewert.

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      OPEN (20, FILE='/tmp/testdatei.txt', STATUS='OLD', ERR=222)

      WRITE (*,*) 'Voller Erfolg'

      CLOSE(20)
      GOTO 333

 222  WRITE(*,*) 'Fehler beim Öffnen der Datei'
 333  END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

15.4.4. CLOSE

Geschlossen wird die Verbindung zur externen Datei mit dem `CLOSE`-Befehl.

CLOSE (liste)

liste:

Element	**Kommentar**
[UNIT =] x	wie bei OPEN
IOSTAT = x	wie bei OPEN
ERR = x	wie bei OPEN
STATUS = x	KEEP ... Datei erhalten DELETE ... Datei löschen

15.4.5. Lesen und Schreiben

Aus Dateien gelesen oder in Dateien geschrieben wird mit den bereits bekannten `READ`- und `WRITE`-Anweisungen.

Element	**Kommentar**
[UNIT =] x	Unit 0 bis 99 bzw. CHARACTER-Variable oder Feld (interne Datei)
[FMT =] x	siehe FORMATIERUNG[1]

1 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20FORTRAN%2077%3A%20EIN-{}%20UND%20AUSGABE%23FORMATIERUNG

Element	Kommentar
REC = x	Datensatznummer bei Direktzugriff (siehe Abschnitt DIREKTZUGRIFF[2])
IOSTAT = x	wie bei READ[3]
ERR = x	Bei Fehler Sprung zur Anweisungsnummer x
END = x	Bei Dateiende Sprung zur Anweisungsnummer x (nicht erlaubt bei Direktzugriff)

Beispiel:

```
0   . | 1    .    2    .    3    .    4    .    5    .    6    .    7 | .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      CHARACTER*80 A

      OPEN (20, FILE='/tmp/testdatei.txt', STATUS='OLD', ERR=222)

 10   CONTINUE
C  Aus Datei lesen
      READ (20, 888, END=20) A
C  Auf Standardausgabe schreiben
      WRITE (*,*) A
      GOTO 10

 20   CLOSE(20)
      GOTO 333

 222  WRITE(*,*) 'Fehler beim Öffnen der Datei'
 888  FORMAT(A)
 333  END
```

2 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20FORTRAN%2077%3A%20EIN-{}%20UND%20AUSGABE%23DIREKTZUGRIFF

3 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20FORTRAN%2077%3A%20EIN-{}%20UND%20AUSGABE%23READ

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

Direktzugriff

OPEN:

Element	Kommentar
ACCESS = x	x ... 'DIRECT'
RECL = x	x ... Datensatzlänge (positive Zahl, ACCESS='DIRECT', in Bytes bzw. bei formatierten Dateien in Characters)

READ/WRITE:

Element	Kommentar
REC = x	x ... Satznummer bei Direktzugriff

Beispiel: Gegeben ist die Textdatei `/tmp/testdatei.txt` mit dem Inhalt

```
Die WRITE-Anweisung dient der Datenausgabe aus einem FORTRAN-Programm
auf ein
externes Gerät. Typisches Beispiel ist die Anzeige von Daten auf dem
Bildschirm.
Formal sieht eine WRITE-Anweisung so aus:
```

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP
```

```
      CHARACTER*10 C

      OPEN (20, FILE='/tmp/testdatei.txt',
     &          STATUS='OLD',
     &          ACCESS='DIRECT',
     &          RECL=10,
     &          ERR=222)

      READ (20, REC=4, ERR=333) C
      WRITE (*,*) C
      READ (20, REC=25, ERR=333) C
      WRITE (*,*) C

      CLOSE (20)
      GOTO 444

 222  WRITE (*,*) 'Fehler beim Öffnen der Datei'
 333  WRITE (*,*) 'Fehler beim Lesen des Datensatzes'
 444  CONTINUE

      END
```

```
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Ausgabe:

```
Datenausga
Fehler beim Lesen des Datensatzes
```

Positionieren bei sequentiellen Dateien

Datensatzzeiger um einen Datensatz zurücksetzen:

```
BACKSPACE ([UNIT=]x [,IOSTAT=y] [,ERR=z])
```

Positionieren an den Dateibeginn:

REWIND ([UNIT=]x [,IOSTAT=y] [,ERR=z])

Schreiben eines Dateiendsatzes:

ENDFILE ([UNIT=]x [,IOSTAT=y] [,ERR=z])

Beispiel:

```
0    .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      CHARACTER*100 C(3)

      OPEN (20, FILE='/tmp/testx.txt',
     &          STATUS='NEW',
     &          ERR=222)

      WRITE (20,*) 'Das ist eine Testdatei'
      WRITE (20,*) 'Dies ist Zeile 2 der Testdatei'
      WRITE (20,*) 'Jenes die Zeile 3 der Testdatei'
      WRITE (20,*) 'Jetzt ist"s aber genug'
      ENDFILE (20, ERR=444)
      REWIND (20, ERR=444)
      READ (20, FMT=555, ERR=333) C
      WRITE (*, FMT=555) C
      BACKSPACE (20, ERR=444)
      READ (20, FMT=555, ERR=333) C(1)
      WRITE (*, FMT=555) C(1)

      GOTO 999

 222  WRITE (*,*) 'Fehler beim Öffnen der Datei'
      GOTO 999

 333  WRITE (*,*) 'Fehler beim Lesen des Datensatzes'
      GOTO 999
 444  WRITE (*,*) 'Sonstiger Fehler'
      GOTO 999
 555  FORMAT (A)
 999  CLOSE (20)

C  Ausgabe:
C   Das ist eine Testdatei
```

```
C   Dies ist Zeile 2 der Testdatei
C   Jenes die Zeile 3 der Testdatei
C   Jenes die Zeile 3 der Testdatei
      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

15.4.6. INQUIRE

Die Anweisung `INQUIRE` dient der Abfrage einiger Eigenschaften von Dateien oder I/O-Units.

INQUIRE (FILE = *x, liste*)

mit x ... Dateiname (inkl. Pfad)

INQUIRE ([UNIT =] *x, liste*)

mit x ... Nummer der I/O-Unit.

liste:

Element	**Kommentar**
ACCESS = x	x: • 'SEQ' ... sequentieller Dateizugriff • 'DIRECT' ... Direktzugriff

Element	Kommentar
BLANK = x	x: • 'NULL' • 'ZERO'
DIRECT = x	x: • 'YES' ... Direktzugriff • 'NO' ... kein Direktzugriff für diese Datei erlaubt • 'UNKNOWN' ... unbekannt
ERR = x	Bei Fehler Sprung zur Anweisungsnummer x
EXIST = x	x: • .TRUE. ... Datei existiert • .FALSE. ... Datei existiert nicht
FORM = x	x: • 'FORMATTED' ... Datei geöffnet für formatierte Datensätze • 'UNFORMATED' ... Datei geöffnet für unformatierte Datensätze

Element	Kommentar
FORMATTED = x	• 'YES' ... formatierte Datensätze sind erlaubt • 'NO' ... formatierte Datensätze sind nicht erlaubt • 'UNKNOWN' ... unbekannt
IOSTAT = x	x ist 0 wenn `INQUIRE` fehlerfrei ausgeführt wurde, ansonsten eine systemabhängige positive Fehlernummer
NAME = x	Der Dateiname wird der Zeichenketten-Variablen x zugewiesen. Hat die Datei keinen Namen, dann ist ist das Ergebnis undefiniert.
NAMED = x	x: • .TRUE. ... Datei besitzt Namen • .FALSE. ... Datei besitzt keinen Namen
NEXTREC = x	x ... Nummer des nächsten Datensatzes
NUMBER = x	x ... Nummer der mit einer externen Datei verbundenen I/O-Unit.

Element	Kommentar
OPENED = x	x: • .TRUE. ... Datei ist geöffnet • .FALSE. ... Datei ist nicht geöffnet
RECL = x	x ... Datensatzlänge bei Direktzugriff
SEQUENTIAL = x	x: • 'YES' ... sequentiell • 'NO' ... nicht sequentiell • 'UNKNOWN' ... unbekannt
UNFORMATTED = x	• 'YES' ... unformatierte Datensätze sind erlaubt • 'NO' ... unformatierte Datensätze sind nicht erlaubt • 'UNKNOWN' ... unbekannt

Beispiel: Datei vorhanden?

```
0    .|  1    .    2    .    3    .    4    .    5    .    6    .    7|   .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      LOGICAL L
```

```
      INQUIRE (FILE='/tmp/testdatei.txt', EXIST=L, ERR=222)
      WRITE (*,*) L
C  Ausgabe:
C    wenn Datei existiert:       T
C    wenn Datei nicht existiert: F

      GOTO 999

 222  WRITE (*,*) 'Fehler!'
 999  CONTINUE
      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Beispiel: Infos zu einer geöffneten Datei

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      LOGICAL EX
      CHARACTER*15, DI, FO, AC, SE
      INTEGER NU

      OPEN (25, FILE='/tmp/testdatei.txt', STATUS='OLD', ERR=222)

      INQUIRE (25, EXIST = EX,
     &             DIRECT = DI,
     &             SEQUENTIAL = SE,
     &             FORMATTED = FO,
     &             ACCESS = AC,
     &             NUMBER = NU,
     &             ERR=222)
      WRITE (*,*) 'EXIST? ', EX
      WRITE (*,*) 'DIRECT? ', DI
      WRITE (*,*) 'SEQUENTIAL? ', SE
      WRITE (*,*) 'FORMATTED? ', FO
      WRITE (*,*) 'ACCESS? ', AC
      WRITE (*,*) 'NUMBER? ', NU
C  Ausgabe, z.B.
C     EXIST?  T
C     DIRECT? YES
C     SEQUENTIAL? YES
C     FORMATTED? YES
```

```
C     ACCESS? SEQUENTIAL
C     NUMBER?           25

      GOTO 999

 222  WRITE (*,*) 'Fehler!'
 999  CLOSE (25)
      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

15.4.7. Interne Dateien

- Interne Dateien sind vom Datentyp CHARACTER (Zeichen oder Zeichenketten)
- Das Lesen aus bzw. das Schreiben in interne Dateien erfolgt immer sequentiell

Beispiel: Schreiben in eine interne Datei

```
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      CHARACTER*15 CH
      REAL R

      R = 12.5678

C  Interne Datei "CH"
      WRITE (CH, *) R

      WRITE (*,*) 'R lexikalisch groesser als Buchstabe "A"? ',
     &            LGE(CH, 'A')
      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .  |  1    .    2    .    3    .    4    .    5    .    6    .    7  |  .    8
```

Beispiel: Lesen aus einer internen Datei

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      CHARACTER*15 CH
      REAL R

      CH = '12.5678'

C  Interne Datei "CH"
      READ (CH, '(F15.5)') R

      WRITE (*,*) 'R    = ', R
      WRITE (*,*) 'R**2 = ', R**2
      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

16. Anhang

16.1. Liste aller FORTRAN 77 Schlüsselwörter

16.2. PAUSE

Die `PAUSE`-Anweisung unterbricht die Programmausführung. Diese Anweisung wurde mit dem Fortran 90/95-Standard aus dem Fortran-Sprachumfang verbannt. Moderne Compiler geben deshalb teilweise bei Verwendung dieser Anweisung eine Warnmeldung (obsolete o.ä.) aus. Das Verfahren zur normalen Fortsetzung des infolge `PAUSE`-Anweisung angehaltenen Programmes ist compilerabhängig, z. B.:

- *gfortran* und *g77*: `go` RETURN
- *g95*: RETURN
- *Intel Fortran Compiler 9.0*: `continue` RETURN

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      WRITE (*,*) 'Hallo '
      PAUSE
      WRITE (*,*) 'Welt!'
      END
```

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .| 1   .   2   .   3   .   4   .   5   .   6   .   7| .   8
```

Ausgabe bei Compilierung mit *gfortran*:

```
 Hallo
PAUSE
To resume execution, type go.  Other input will terminate the job.
```

Ausgabe bei Compilierung mit *g95*:

```
 Hallo
PAUSE statement executed.  Hit Return to continue
```

Ausgabe bei Compilierung mit *ifort*:

```
 Hallo
FORTRAN PAUSE
PAUSE prompt>
```

16.3. INCLUDE

`INCLUDE` ermöglicht das Einbinden einer Datei. `INCLUDE` ist nicht explizit im FORTRAN 77-Standard angeführt, sondern wurde 1978 als Erweiterung des FORTRAN-Standards vom US-amerikanischen DoD im MilStd 1753 festgelegt (siehe auch Abschnitt MILSTD 1753[1]).

Beispiel:

1 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20FORTRAN%2077%3A%20ANHANG%23MILSTD%201753%2C%20DOD%

Datei *bsp.f*:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      INCLUDE 'inc.f'

      WRITE(*,*) 'Fläche ist', 10**2*PI, 'cm2'
C  Ausgabe: Fläche ist   314.1593     cm2

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

Datei *inc.f*:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      REAL PI
      PARAMETER(PI=3.141593)


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

16.4. Hollerith-Konstanten

Eine veraltete Möglichkeit Zeichenketten anzugeben stellen die Hollerith[2]-Konstanten dar. Eine Hollerith-Konstante besteht aus

---

20Supplement%20To%20American%20National%20Standard%20X3.9-{}1978

2 Herman Hollerith (* 1860, † 1929), Erfinder des Hollerith-Lochkartenverfahrens, Herman Hollerith ^{http://de.wikipedia.org/wiki/Herman%20Hollerith}

1. Einer positiven Ganzzahl, welche die Zeichenkettenlänge angibt
2. Dem Buchstaben H
3. Der Zeichenkette

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      CHARACTER*5 C
      DATA C /5HUralt/

      WRITE (*,*) C
 C  Ausgabe: Uralt

      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|  .    8
```

16.5. Arithmetisches IF

Schon in grauer Vorzeit fanden Programmierer arithmetische IF-Anweisungen sehr unterhaltsam, weil damit der Programmcode interessanter gestaltet werden konnte (QUELLE: ED POST - REAL PROGRAMMERS DON'T USE PASCAL[3]). Diese Aussage ist begreiflicherweise auch heute noch uneingeschränkt gültig.

IF(ausdruck) ziel1, ziel2, ziel3

mit

3 HTTP://WWW.EE.RYERSON.CA/~{}ELF/HACK/REALMEN.HTML

ausdruck	springe zu
< 0	ziel1
= 0	ziel2
> 0	ziel3

Beispiel:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

       PROGRAM BSP

       INTEGER I

       I = -55

       IF(I) 100, 200, 300
       GOTO 999

  100  WRITE(*,*) "Negative Zahl"
 C Ausgabe: Negative Zahl
       GOTO 999

  200  WRITE(*,*) "Null"
       GOTO 999

  300  WRITE(*,*) "Positive Zahl"

  999  CONTINUE
       END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

16.6. ASSIGN und Assigned GOTO

Noch wesentlich amüsanter als bei der unachtsamen Verwendung des arithmetischen IF können die Ergebnisse durch Verwendung von Assign-Anweisungen ausfallen.

ASSIGN weist eine Zielmarke einer INTEGER-Variablen zu:

ASSIGN zielmarke TO variable

Verwendung finden kann dies neben anderem beim Assigned GO-TO:

GOTO variable (liste)

Beispiel:

```
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|   .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

      PROGRAM BSP

      INTEGER VAR

C ...
      ASSIGN 200 TO VAR
C ...
C ...
C ...

      GOTO VAR (100, 200, 300)

 100  WRITE(*,*) "Negative Zahl"
      GOTO 999

 200  WRITE(*,*) "Null"
C Ausgabe: Null
      GOTO 999

 300  WRITE(*,*) "Positive Zahl"
 999  CONTINUE
      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   .|  1    .    2    .    3    .    4    .    5    .    6    .    7|   .    8
```

16.7. MilStd 1753, DoD Supplement To American National Standard X3.9-1978

Nachfolgend stichwortartig die vom amerikanischen Department of Defense im Jahr 1978 ergänzend zu FORTRAN 77 geforderten Spracheigenschaften[4]

- END DO (gelabelt, für DO-Schleifen)
- DO WHILE-Schleife
- INCLUDE
- IMPLICIT NONE
- Manipulation von Bitfeldern (IOR, IAND,ISHFT, ...)
- READ/WRITE-Verhalten nach EOF.

<references />

4 http://www.fortran.com/fortran/mil_std_1753.html

Teil IV.

Fortran 95

17. Programmaufbau

17.1. Beispiel: Hallo Welt

Fortran 90/95-Code (free source form)

```
program hallo
  ! Das typische "Hallo Welt"-Programm
  write( *, * ) 'Hallo Welt!'
end program hallo
```

Dieses Programm besteht aus den gleichen Anweisungen wie ein entsprechendes FORTRAN 77-Programm und wurde in der sogenannten "free source form" verfasst. Aus diesem Grund ist die erste Spalte nicht mehr dem Kommentarzeichen vorbehalten. Kommentare werden hier durch ein Rufzeichen eingeleitet. Alles rechts vom Rufzeichen wird als Kommentar behandelt. Ein Kommentar kann also auch nach einer Anweisung stehen. Im Gegensatz zu einem FORTRAN 77-Programm wird in Fortran 90/95 oft die Kleinschreibung bevorzugt. Dies ist aber nicht zwingend erforderlich, da die Fortran-Compiler case-insensitiv sind. D.h. sie unterscheiden nicht zwischen Groß- und Kleinschreibung. Eine Einrückung von Blöcken ist nicht zwingend erforderlich, fördert aber die Übersichtlichkeit des Programmcodes.

Die Anweisung in der ersten Zeile kennzeichnet die Programmeinheit als Hauptprogramm und gibt ihr die Bezeichnung `hallo`. Es folgt eine Kommentarzeile. Dann folgt die Anweisung, einen

String auf die Standardausgabe zu schreiben. Und schließlich signalisiert die `end`-Anweisung das Programmende.

17.2. Das Zeilenformat

Fortran 90/95 bietet zwei verschiedene Programmaufbaumöglichkeiten:

- free source form
- fixed source form

Die "free source form" ist neu in Fortran 90/95. Sie bietet die Möglichkeit Programme ohne fixe Spaltenzuordnung zu schreiben, wie dies auch in den meisten anderen gebräuchlichen Programmiersprachen üblich ist. Zusätzlich ist in Fortran 90/95 aus Kompatibilitätsgründen auch das alte FORTRAN 77-Zeilenformat (fixed source form) enthalten. Dieses sollte in neuen Programmen aber nicht mehr verwendet werden.

Normalerweise gilt, dass jede Fortran-Anweisung in einer eigenen Zeile steht. Bei Verwendung der "free source form" gelten folgende Bedingungen. Eine Zeile darf maximal 132 Zeichen lang sein. Als Zeilenumbruchzeichen dient das Kaufmanns-Und (&). Das Kaufmanns-Und steht in diesem Fall immer am Ende der fortzuführenden Zeile, optional auch zusätzlich am Beginn der Fortsetzungszeile. Standardmäßig sind maximal 40 Fortsetzungszeilen erlaubt.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  ! Leerzeilen werden vom Compiler ignoriert
  ! Vereinbarungsteil
  implicit none               ! implizite Datentypvereinbarung
 ausschalten
  integer       :: a, b, c    ! Variablendeklaration (Ganzzahlen)
  character(25) :: str        ! Variablendeklaration (String mit einer
```

```
 Länge von 25 Zeichen)

  ! Aktionsteil
  a = 5
  b = 7
  c = a + &
  ! und jetzt kommt eine Fortsetzungszeile
  b
  write( *, * ) c

  ! auch Strings oder Schlüsselwörter können auf der nächsten Zeile
 fortgesetzt werden.
  ! Dabei ist unbedingt darauf zu achten, dass die Fortsetzungszeile
 auch mit einem
  ! Kaufmanns-Und eingeleitet wird, da ansonsten einige Compiler
 Fehlermeldungen liefern
  str = "Hal&
  &lo Welt!"

  wr&
  &ite( *, * ) str
! Ausgabe:
!   12
!   Hallo Welt!
end program bsp
```

Mehrere Anweisungen können in eine Zeile geschrieben werden, wenn eine Trennung durch jeweils ein Semikolon erfolgt.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
  integer :: a, b, c
  a = 5; b = 7; c = a + b
  write( *, * ) c
! Ausgabe:
!   12
end program bsp
```

17.3. Die Programmstruktur für das Hauptprogramm

Ein Hauptprogramm weist in einfachster Darstellung immer folgende Struktur auf:

1. `program` *Name*
2. Vereinbarungsteil
3. Aktionsteil
4. `end` [`program` [*Name*]]

Die in eckigen Klammern angegebenen Bestandteile sind optional. Sie sollten dennoch stets mit angegeben werden. Genau eine `end`-Anweisung (mit oder ohne optionale Bestandteile) ist obligatorisch. Bei Hauptprogramm und Unterprogrammen ist `end` eine ausführbare Anweisung. Ein Fortran-Programm darf genau ein Hauptprogramm enthalten, es ist der Startbereich für die Programmausführung.

17.4. Der Fortran-Zeichenvorrat

Fortran 95-Programme bestehen standardmäßig aus folgenden Zeichen:

- Großbuchstaben: A bis Z
- Kleinbuchstaben: a bis z
- Ziffern: 0 bis 9
- Den 13 FORTRAN 77-Sonderzeichen: + - * / = ( ) : , . ’ $ und dem Leerzeichen
- Unterstrich (Underscore): _
- Weitere Sonderzeichen: ! ? " & ; < >

Ein Fortran 90/95-Compiler ist case-insensitiv: er unterscheidet **nicht** zwischen Groß- und Kleinbuchstaben. Stringkonstanten können natürlich alle ASCII-Zeichen beinhalten.

17.5. Symbolische Namen

Standardmäßig dürfen symbolische Namen maximal 31 Zeichen lang sein. Das erste Zeichen muss immer ein Buchstabe sein. Anschließend sind alphanumerische Zeichen (Buchstabe oder Ziffer) und Unterstriche erlaubt. Im Gegensatz zu FORTRAN 77 dürfen nun keine Leerzeichen innerhalb eines symbolischen Namens auftreten.

17.6. Reservierte Schlüsselwörter?

Im Fortran 95-Standard ist zwar die Rede von Schlüsselwörtern (statement keywords), wie z.B. `if, do, real, write`. Allerdings wird eindeutig darauf hingewiesen, dass diese nicht reserviert sind. Das heißt, solche Schlüsselwörter können auch für eigene Bezeichner verwendet werden.

Beispiel: Folgender Programmcode ist in Fortran 90/95 gültig, aber nicht empfehlenswert

Fortran 90/95-Code (free source form)

```
program bsp
  implicit none
  integer :: integer, write, if
  integer = 6
  write = 5
  if = 1

  write( *, * ) integer + write + if
```

```
end program bsp
```

17.7. Details zur Anordnungsreihenfolge von Anweisungen

Fortran schreibt eine gewisse Anordnungsstruktur der einzelnen Programmelemente vor. So ist das folgende Programm

Fortran 90/95-Code (free source form)

```
program hallo
  write( *, * ) "Hallo Welt"
  implicit none
end program hallo
```

nicht standardkonform, da die `implicit none`-Vereinbarung nicht nach einer ausführbaren Anweisung folgen darf. So müsste es richtig lauten

Fortran 90/95-Code (free source form)

```
program hallo
  implicit none
  write( *, * ) "Hallo Welt"
end program hallo
```

Im Fortran 90/95-Standard ist genau festgelegt wann bestimmte Programmelemente auftreten dürfen. Prinzipiell gilt, dass Vereinbarungs-Anweisungen (nichtausführbare Anweisungen) vor Aktions-Anweisungen (ausführbaren Anweisungen) stehen. Doch es gibt Ausnahmen, z.B. darf die nichtexekutierbare `format`-Anweisung auch zwischen oder nach ausführbaren Anweisungen stehen. Eindeutig ist Folgendes geregelt. Die Programmeinheit beginnt mit dem charakteristischen Schlüsselwort, z.B. `function`. Werden Module eingebunden (`use ...`), so erfolgt dies

vor jeder anderen Vereinbarung oder ausführbaren Anweisung. Danach folgt ein "Mischbereich" (im Fortran-Standard wird das natürlich nicht "Mischbereich" genannt). Vor dem `end`-Statement darf ein `contains`-Abschnitt mit Unterprogrammen stehen.

Fortran 90/95-Code (free source form)

```
program (, function, subroutine, ...) ...
  use ...

  ! Hier folgt nun der so genannte "Mischbereich"

  contains
    interne Unterprogramme oder Modul-Unterprogramme
end ...
```

Im "Mischbereich" ist die Anordnung einzelner Elemente zwar auch im Standard geregelt, jedoch gibt es Elemente die an jeder Stelle des Mischbereichs auftreten dürfen, bei anderen ist wiederum klar festgelegt , dass sie nur nach oder vor anderen Elementen auftreten dürfen. Hier soll deshalb nur eine vereinfachte Variante gezeigt werden. Für genauere Informationen wird auf den Fortran-Working-Draft verwiesen.

Fortran 90/95-Code (free source form)

```
program (, function, subroutine, ...) ...
  use ...

! --------- Beginn "Mischbereich" (mögliche Anordnungsreihenfolge)
---------
  implicit ...

  ! Definiton von Datenverbunden, Interface-Blöcke, benannte
Konstanten, Variablendeklarationen

  ! ausführbare Anweisungen

   ... format ...
! --------- Ende "Mischbereich"
-------------------------------------------

  contains
```

```
    Interne Unterprogramme bzw. Modul-Unterprogramme
end ...
```

Nicht jede Anweisung ist in jeder Programmeinheit erlaubt. So dürfen z.B. keine `format`-Anweisungen im Hauptteil eines Moduls (wohl aber in Modul-Unterprogrammen) verwendet werden.

18. Datentypen, Variablen, Wertzuweisungen, Konstanten

Dieses Kapitel handelt von Datentypen, Variablen, Konstanten und der Wertzuweisung.

18.1. Datentypen

18.1.1. Arithmetische Datentypen

Datentyp	**Kommentar**	**Beispiele (Konstanten)**
`integer`	Ganzzahlen	15, -6500, 200000000
`real`	Gleitkommazahlen einfacher Genauigkeit	3.1415, -5.5, .7e3, 12.5E-5
(`double precision`)	Gleitkommazahlen doppelter Genauigkeit (aus FORTRAN 77)	3.1415D0, -5.5D0, .7d3, 12.5D-5

Datentyp	**Kommentar**	**Beispiele (Konstanten)**
`complex`	Komplexe Zahlen (zwei `real`-Zahlen)	(3.1415, -5.5), (1.4, 7.1E4)

`double precision` ist in Fortran 95 nur noch aus historischen Gründen (FORTRAN 77) vorhanden. Fortran 95 bietet für Datentypen höherer Genauigkeit bzw. mit größerem Zahlenbereich andere Sprachmittel. Diese Thematik wird später erläutert (DATENTYPEN HÖHERER GENAUIGKEIT[1]).

Binär-, Oktal- oder Hexadezimalzahlen können als "boz literal constants" angegeben werden:

Schreibweise 1	**Schreibweise 2**	**Kommentar**
B"*zahl*"	B'*zahl*'	**B**inäre Zahl
O"*zahl*"	O'*zahl*'	**O**ktalzahl
Z"*zahl*"	Z'*zahl*'	Hexade**z**imalzahl

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer :: a

  a = Z"AB1C"
  write(*,*) a
  ! Ausgabe: 43804

  a = O"7134"
  write(*,*) a
  ! Ausgabe: 3676

  a = B'101110110001'
```

1 Kapitel 25 auf Seite 217

```
  write(*,*) a
  ! Ausgabe: 2993
end program bsp
```

18.1.2. Logischer Datentyp

Datentyp	Kommentar	Mögliche Werte
logical	Logischer Datentyp (wahr oder falsch)	.TRUE., .FALSE.

18.1.3. Zeichenketten

Datentyp	Kommentar	Beispiel (Konstante)
character(n)	Zeichenkette (String) mit einer Länge von n Zeichen	'Hallo, Welt!', "Hallo, Welt!"
character(len=n)	-"-	
character*n	-"- (FORTRAN 77-Stil, sollte nicht mehr verwendet werden)	
character	Zeichenkette (String) mit einer Länge von einem Zeichen	'H', "h"

Beachte: Zeichenketten können in Fortran 95 in Apostrophe oder Anführungszeichen eingeschlossen werden.

Beispiel:

```
"Wie geht's?"
```

```
'Er sagte: "Hallo"'
```

18.2. Variablen

Eine Variable ist charakterisiert durch einen

- symbolischen Namen
- Datentyp
- Wert
- Speicherplatz

Beim Programmstart hat eine Variable keinen definierten Wert. Eine Variable kann ihren Datentyp auf zwei Arten erhalten, durch implizite oder explizite Typanweisung.

18.2.1. Implizite Typanweisung

Bei der impliziten Typanweisung bestimmt der Anfangsbuchstabe des Variablenbezeichners den Datentyp.

Datentyp	**Anfangsbuchstabe der Variablen**
integer	Buchstaben I bis N (oder i bis n)
real	restliche Buchstaben

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  b1 = 8.9
  c1 = 3.
  i1 = B1/C1

  write (*,*) i1
  ! Das Ergebnis ist 2, da i1 implizit als integer definiert ist
end program bsp
```

Die Standardzuordnung der Anfangsbuchstaben kann durch das Schlüsselwort `implicit` auch geändert werden. Infolge der durch implizite Typanweisung entstehenden Fehlermöglichkeiten ist es sinnvoll, die implizite Typanweisung komplett auszuschalten. Dies wird durch die Anweisung

```
implicit none
```

gleich nach der `program`-Anweisung erreicht. Dann muss der Datentyp jeder Variablen explizit festgelegt werden.

18.2.2. Explizite Typanweisung

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  real :: b
  real :: c
  real :: i
  ! alternativ auch als real :: b, c, i

  b = 8.9
  c = 3.
  i = b/c

  write (*,*) i
  ! Das Ergebnis ist 2.966666
```

```
end program bsp
```

18.3. Benannte Konstanten

Benannte Konstanten werden in Fortran 95 folgendermaßen festgelegt:

datentyp, parameter :: *symname* = *wert*

Beispiele:

```
real, parameter:: PI = 3.1415, PIFAC = PI/2.0
```

```
character(5), parameter :: str = 'Hallo'
```

Der zugewiesene Wert kann eine Konstante (Literal) oder eine schon definierte benannte Konstante sein.

Für die Zeichenkettenlänge ist bei benannten Konstanten auch eine *-Schreibweise möglich. Dies erspart die explizite Angabe der Stringlänge.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  character(*), parameter :: a = 'Jetzt wird auch die Stringlaenge
 festgelegt'

  write(*,*) a
  ! Ausgabe: Jetzt wird auch die Stringlaenge festgelegt
end program bsp
```

18.4. Wertzuweisung

Wertzuweisungen haben wir schon kennengelernt:

```
variable = ausdruck
```

Beispiel:

```
k = 1
k = k + 2
```

Die Wertzuweisung an eine Variable ist, wie am vorigen und auch am nächsten Beispiel zu ersehen, nicht zu verwechseln mit einer mathematischen Gleichung. Der Ausdruck

```
k + 2 = 5
```

wäre zwar mathematisch korrekt. Als Wertzuweisung in einem Fortran-Programm ist dies aber keine mögliche Formulierung. `k+2` ist kein zulässiger Ausdruck auf der linken Seite des Zuweisungsoperators (L-Wert).

In Fortran ist auch keine Kette von Wertzuweisungen möglich. Der folgende Ausdruck ist **nicht** erlaubt und liefert eine Fehlermeldung.

```
i = j = k = 1.5
! Fehler
```

19. Felder

19.1. Beispiel

Fortran 90/95-Code (free source form)

```
program bsp
  implicit none

  real, dimension(10) :: arr
  ! ACHTUNG! Array startet mit dem Index 1
  ! arr(0) waere ein Fehler!

  arr(1) = 1.5
  arr(2) = 2.5
  arr(10) = 10.5

  write(*,*) arr(1)
  ! 1.500000 wird ausgegeben

  write(*,*) arr(10)
  !  10.500000 wird ausgegeben
end program bsp
```

19.2. Eindimensionale Felder

19.2.1. Statische Speicherallokation

Variante 1: Die `dimension`-Anweisung

```
real, dimension(10) :: arr
```

Der Feldindex läuft von 1 bis 10.

Variante 2: Einfach

```
real :: var(10)
```

Variante 3: Verwendung von benannten Konstanten

```
integer, parameter :: MAXIND = 10
real, dimension(MAXIND) :: arr
```

Hier erfolgt die Festlegung der Feldgröße über eine benannte Konstante.

Variante 4: Explizite Angabe der Feldgrenzen

```
real, dimension(0:9) :: arr
```

Hier wird Unter- und Obergrenze explizit angegeben. Der Index läuft nun von 0 bis 9. Die Feldgrenzen können auch negativ sein.

19.2.2. Dynamische Speicherallokation

Mit Fortran 90/95 kann der benötigte Speicherplatz für ein Feld dynamisch angefordert werden. Zu diesem Zweck gibt es das Schlüsselwort `allocatable`, welches unmittelbar bei der Felddeklaration die beabsichtigte dynamische Speicherallokation signalisiert.

datentyp, dimension(:), allocatable :: *variablenbezeichner*

Die intrinsische Funktion `allocate` dient dann der konkreten Speicherallokation im Aktionsteil des Programmes, also u.a. der Festlegung der konkreten Feldgröße. Der reservierte Speicherplatz kann mit der Funktion `deallocate` auch wieder freigegeben werden.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  real, dimension(:), allocatable :: arr   ! dynamisches
 eindimensionales Feld

  integer :: status

  ! Allokation
  allocate(arr(0:9), stat=status)

  ! Feld mit Werten belegen,
  ! ....
  ! ....

  ! Deallokation
  deallocate(arr, stat=status)
end program bsp
```

Die Statusvariable sollte nach Ausführung der Allokationsfunktionen jeweils den Wert 0 aufweisen. Andere Werte stehen für eine Fehlermeldung.

Die Funktion `allocated()` hilft bei der Überprüfung des Allokationsstatus eines Feldes. Ist für das abgefragte Feld bereits Speicherplatz allokiert, dann liefert die Funktion `.TRUE.`, ansonsten `.FALSE.`

```
! ...
if (.not. allocated(arr)) then
  allocate(arr(groesse), stat=s)
end if
! ...
```

Weitere Funktionen bezüglich Felder sind im Kapitel STANDARDFUNKTIONEN[1] aufgelistet.

19.3. Mehrdimensionale Felder

Für mehrdimensionale Felder gelten die gleichen Varianten wie für eindimensionale Felder. Die Speicherreihenfolge ist spaltenorientiert. Das bedeutet, der erste Index variiert am schnellsten: $a_{11}, a_{21}, \ldots, a_{(n-1)m}, a_{nm}$

1 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN_95%3A_STANDARDFUNKTIONEN%23FELDFUNKTIONEN

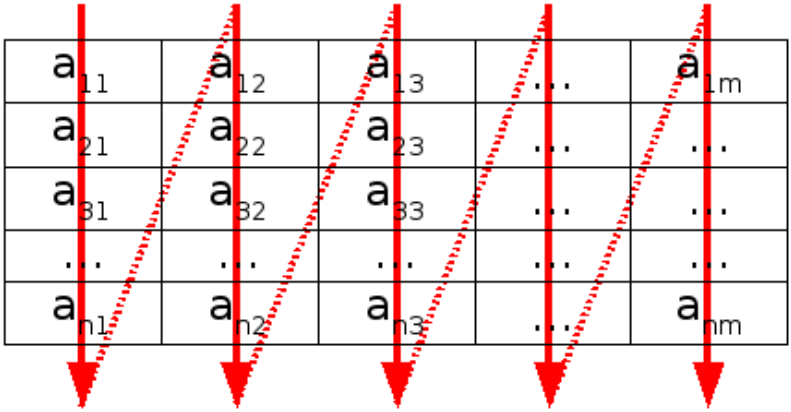


Abb. 20

Ein mehrdimensionales Feld kann auch in Fortran 90/95 maximal 7-dimensional gestaltet werden.

19.3.1. Beispiel: Ein 2-dimensionales Feld

Fortran 90/95-Code (free source form)

```
program bsp
  implicit none

  character(10), dimension(0:9, 2:5) :: arr

  arr(0, 2) = 'Hallo'
  arr(1, 2) = 'Welt'
  !  ...
  arr(9, 5) = 'Universum'

  write (*,*) arr(0, 2)
  ! Ausgabe:  Hallo

  write (*,*) arr(9, 5)
```

```
  !  Ausgabe: Universum
end program bsp
```

19.3.2. Beispiel: Spaltenorientierte Speicherreihenfolge

Die 3x3-Matrix $A = \begin{pmatrix} 1 & -5 & 0 \\ 40 & 3 & -2 \\ -1 & 9 & 65 \end{pmatrix}$ soll in ein Fortran-Programm eingelesen und wieder komplett ausgegeben werden. Zusätzlich soll auch der Wert des Feldelementes a_{23} (2. Zeile, 3.Spalte, Wert=-2) separat ausgegeben werden.

Fortran 90/95-Code (free source form)

```
program bsp
  implicit none
  integer, dimension(3,3) :: arr(3,3)
! Feldelemente einlesen
  write (*,*) 'Werte (spaltenorientierte Eingabe):'
  read (*,*) arr

! Komplettes Feld ausgeben
  write (*,*) 'Gesamtfeld = ' , arr
!  a23 ausgeben
  write (*,*) 'a23 = ', ARR(2,3)
end program bsp
```

Ein-/Ausgabe:

```
 Werte (spaltenorientierte Eingabe):
1
40
-1
-5
3
9
0
-2
```

```
65
 Gesamtfeld =             1          40          -1          -5
   3          9           0
     -2          65
 a23 =           -2
```

19.4. Feldinitialisierung

Felder lassen sich auch gleich bei der Deklaration mit Werten initialisieren.

19.4.1. Array constructor

Bei eindimensionalen Feldern kann die Feldinitialisierung direkt mittels *array constructor* erfolgen.

Beispiel:

```
real, dimension(5) :: arr = (/1.1, 1.2, 1.3, 2.1, 2.4/)
```

19.4.2. Reshape

Für mehrdimensionale Felder besteht die Möglichkeit der direkten Verwendung des *array constructors* nicht. Stattdessen kann eine derartige Initialisierung über den Umweg der `reshape`-Funktion geschehen.

Beispiel: Die Matrix

```
1 2 3
1 2 3
```

soll in einem 2D-Feld gespeichert werden.

```
integer, dimension(2,3) :: arr = reshape( (/1, 1, 2, 2, 3, 3/), (/2,
3/) )
```

19.5. Teilfelder

Ähnlich wie das schon in FORTRAN 77 bei Zeichenketten möglich war, können nun auch Teilfelder direkt angesprochen werden.

Prinzip	**Beschreibung**
feldname (*anfang*:*ende*)	von *anfang* bis *ende*
feldname (:*ende*)	vom 1. Feldelement bis *ende*
feldname (*anfang*:)	von *anfang* bis zum letzten Feldelement

- *anfang* >= untere Indexgrenze
- *ende* <= obere Indexgrenze

Beispiel: Eindimensionale Felder **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  character(2), dimension(5)    :: arr = (/'A1', 'A2', 'A3', 'A4',
 'A5'/)
  integer,      dimension(-3:1) :: i   = (/-30, -20, -10, 0, 10/)
```

```
  write (*,*) arr(2:4)
  ! Ausgabe: A2A3A4

  write (*,*) arr(2:)
  ! Ausgabe: A2A3A4A5

  write (*,*) arr(:3)
  ! Ausgabe: A1A2A3

  write (*,*) arr(1)
  ! Ausgabe: A1

  write (*,*) arr
  ! Ausgabe: A1A2A3A4A5

  write (*,*) i(-2:0)
  ! Ausgabe: -20   -10    0
end program bsp
```

20. Arithmetische Ausdrücke

20.1. Arithmetische Operatoren

Fortran 95 kennt wie FORTRAN 77 folgende arithmetische Operatoren

Operator	Kommentar
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
**	Exponentiation

20.1.1. Operatorenpriorität

Die Prioriät der arithmetischen Operatoren entspricht den mathematischen Konventionen.

- Klammerung vor allem anderen, z.B. `(a+b)*c` ⇔ `a*c+b*c`
- Exponentiation vor Punktrechnung, z.B. `a*b**c` ⇔ `a*(b**c)`
- Punktrechnung vor Strichrechnung, z.B. `a+b*c` ⇔ `a+(b*c)`

20.1.2. Berechnungsfolge bei gleicher Priorität

- Klammerung, Punktrechnung und Strichrechnung: →
  Beispiel: `a*b/c*d`⇔ `((a*b)/c)*d`

- Exponentiation: ←
  Beispiel: `a**b**c`⇔ `a**(b**c)`

Außerdem ist zu beachten, dass niemals zwei Operatoren direkt aufeinander folgen dürfen.

Beispiel: Der Ausdruck `1.5**-1` ist in Fortran 95 falsch und führt zu einer Fehlermeldung. Richtig ist `1.5**(-1)`

20.2. Ergebnisdatentyp

20.2.1. Operanden gleichen Datentyps

Bei Operanden gleichen Datentyps erhält das Ergebnis den Datentyp der Operanden.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  real :: a

  a = 3/2
  !  3 ist ein integer und 2 ist auch ein integer,
  !  daher muss das Ergebnis auch ein integer sein, also 1.
  !  Die Zuweisung an die real-Variable a stellt das
  !  Ergebnis nicht mehr richtig.

  write(*,*) a
  ! Ausgabe: 1.00000
end program bsp
```

20.2.2. Implizite Typumwandlung bei Operanden gemischten Datentyps

Weisen die Operanden unterschiedliche Datentypen auf, so wird bei jeder Operation, falls nötig, das Ergebnis dem höherwertigen Datentyp angepasst.

integer→ real→ complex

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  real :: a

  a = 3/2.
  !  2. ist ein real. Jetzt stimmt das Ergebnis.

  write (*,*) a
  ! Ausgabe: 1.500000
end program bsp
```

20.2.3. Explizite Typumwandlung

Fortran 95 besitzt auch Funktionen zur expliziten Umwandlung des Datentyps. Diese werden im Kapitel STANDARDFUNKTIONEN[1] näher beschrieben.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  real :: r
```

1 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN_95%3A%20STANDARDFUNKTIONEN%20

```
  complex :: c

  r = 2.
  c = cmplx(r)

  write (*,*) c
  ! Ausgabe: ( 2.000000 , 0.000000 )
end program bsp
```

21. Logische Ausdrücke

Logische Ausdrücke können zwei Zustände annehmen, `.TRUE.` oder `.FALSE.`.

21.1. Logische Operatoren

Operator	**Kommentar**
.NOT.	logisches NICHT
.AND.	logisches UND
.OR.	logisches ODER
.EQV.	logische Äquivalenz
.NEQV.	logische Antivalenz

21.2. Wahrheitstafel

a	**b**	**.NOT. a**	**a .AND. b**	**a .OR. b**	**a .EQV. b**	**a .NEQV. b**
.TRUE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.
.TRUE.	.FALSE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	.FALSE.	.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  logical :: l

  l = .TRUE.

  write(*,*) .NOT. l
  ! Ausgabe: F
end program bsp
```

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  logical :: a, b

  a = .TRUE.
  b = .FALSE.

  write (*,*) a .NEQV. b
  ! Ausgabe: T
end program bsp
```

21.3. Operatorenpriorität

1. Klammerung () bindet am stärksten
2. `.NOT.`
3. `.AND.`
4. `.OR.`
5. `.EQV.`, bzw. `.NEQV.`

22. Vergleichsausdrücke

22.1. Vergleichsoperatoren

Zum Vergleichen zweier arithmetischer Ausdrücke oder von Strings gibt es Vergleichsoperatoren. Das Ergebnis eines Vergleichs ist ein logischer Wert (`.TRUE.` oder `.FALSE.`).

Operator in Fortran 95	**Operator in FORTRAN 77**	**Kommentar**
<	.LT.	less than (kleiner als, <)
<=	.LE.	less equal (kleiner gleich, <=)
>	.GT.	greater than (größer als, >)
>=	.GE.	greater equal (größer gleich, >=)
==	.EQ.	equal (gleich, ==)
/=	.NE.	not equal (ungleich, !=)

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer :: a, b
```

```
  a = 5
  b = 6

  write (*,*) A < B
  ! Ausgabe: T
end program bsp
```

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  character(len=5) :: a, b

  a = "Halli"
  b = "Hallo"

  write (*,*) a < b
  ! Ausgabe: T
end program bsp
```

22.2. Operatorenpriorität

1. Klammerung
2. Arithmetische Operatoren
3. Vergleichsoperatoren
4. Logische Operatoren

23. Stringoperationen

23.1. Verknüpfungsoperator

Operator	Kommentar
//	Operator zum Verknüpfen von Strings

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  character(len=4) :: a
  character(len=10) :: b

  a = 'How '
  b = 'do you do.'

  write(*,*) a // b
  ! Ausgabe: How do you do.
end program bsp
```

23.2. Teilketten

Ein String ist ein `character`-Feld. Auf die Stringelemente kann wie in einem Feld zugegriffen werden.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  character(10) :: a

  a='Hallo Welt'

  write(*,*) a(2:4)
  ! Ausgabe: all

  write(*,*) A(5:)
  !  Ausgabe: o Welt

  write (*,*) A(:3)
  !  Ausgabe: Hal
end program bsp
```

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  character(10) :: a

  a='Hallo Welt'
  a(7:) = 'XYZ'

  write(*,*) a
  !  Ausgabe: Hallo XYZ
end program bsp
```

24. Verzweigungen und Schleifen

24.1. Einleitung

Neben der sequentiellen Programmausführung bietet Fortran 90/95 andere grundlegende Kontrollstrukturen zur Steuerung des Programmablaufs.

```
do




stop
exit
cycle
```

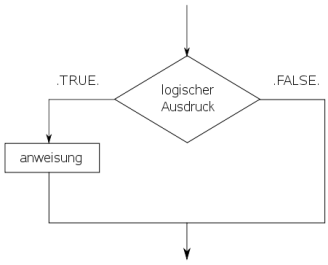
Zwecks komfortabler Manipulation von Feldinhalten stehen zusätzliche Spezialkonstrukte zur Verfügung:

All diese Programmelemente werden in diesem Kapitel detailliert dargestellt. Neben den genannten Steuerkonstrukten sind in Fortran 90/95 auch noch das berüchtigte `goto` und einige als veraltet gekennzeichnete Sprunganweisungen möglich. Fortran 90 und Fortran 95 unterscheiden sich in Details bei Schleifen und

Verzweigungen. Da Fortran 90 durch Fortran 95 bzw. beide durch den Fortran 2003-Standard abgelöst wurden, wird auf die Unterschiede nicht näher eingegangen.

24.2. if-Verzweigungen

24.2.1. Der if-Einzeiler

Form	Strukturbild
if(logischer ausdruck) anweisung	 Abb. 24

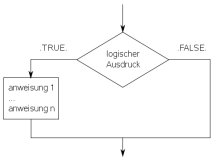
Der if-Einzeiler oder das "if statement" ist die einfachste Form, um in Fortran-Programmen eine Verzweigung zu realisieren. Ist der angegebene logische Ausdruck wahr, dann wird die Anweisung ausgeführt, sonst nach dem if-Einzeiler mit der Programmausführung fortgesetzt. Dieses Konstrukt kann nur eine einzelne Anweisung verdauen! Die gesamte if-Struktur inklusive Anweisung muss in einer Zeile angegeben werden. Eine konventionelle Zeilenfortsetzung mit dem &-Zeichen ist aber möglich. Einige Anweisungen sind nicht erlaubt. So kann die Anweisung nicht wieder ein "if statement" sein (geschachtelte if-Einzeiler). Ebenso darf der

Anweisungsteil nicht aus `end program`, `end function` oder `end subroutine` bestehen.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer :: i

  i = 2
! Der logische Ausdruck ist .true. , "Hallo" wird ausgegeben
  if( i == 2 ) write( *, * ) 'Hallo'
! Das funktioniert nicht und ist deshalb auskommentiert
!  if( i == 2 )
!    write( *, * ) 'Hallo'
! So geht's aber
  if( i == 2 )             &
    write( *, * ) 'Hallo'
! Folgendes wiederum wird nicht das ev. erhoffte Resultat zeigen (=
 keine
! Ausgabe, da der logische Ausdruck .false.) Die durch Strichpunkt
! abgetrennte write-Anweisung gehört nicht mehr zum "if statement",
! sondern ist schon eine eigenständige Anweisung ausserhalb des
! if-statements
  if( i == 3 ) write( *, * ) 'Hallo 1'; write( *, * ) 'Hallo 2'
! Ausgabe:
!   Hallo
!   Hallo
!   Hallo 2
end program bsp
```

24.2.2. if-then

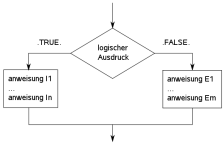
Kurzform	Langform	Strukturbild
if(logischer ausdruck) then anweisungsblock end if	name: if(logischer ausdruck) then anweisungsblock end if name	 Abb. 25

Das if-then-Konstrukt erlaubt eine konventionelle einseitige Verzweigung. Im Gegensatz zum if-Einzeiler ist hier ein Anweisungsblock mit beliebig vielen Anweisungen erlaubt. Einem if-then-Konstrukt, wie auch den meisten nachfolgenden Verzweigungs- und Schleifentypen, kann eine Bezeichnung mitgegeben werden.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer :: i

  i = 2

  if( i == 2 ) then
    write( *, * ) 'Hallo1'
  end if
  checkit: if( i /= 3 ) then
    write( *, * ) 'Hallo2'
  end if checkit
! Ausgabe:
!   Hallo 1
!   Hallo 2
end program bsp
```

24.2.3. if-then-else

Kurzform	Langform	Strukturbild
if(logischer ausdruck) then if-anweisungsblock else else-anweisungsblock end if	name: if(logischer ausdruck) then if-anweisungsblock else name else-anweisungsblock end if name	 Abb. 26

Dies ist eine typische zweiseitige Verzweigung. Je nachdem, ob die Bedingung zu wahr oder falsch ausgewertet wird, wird der if- oder der else-Anweisungsblock ausgeführt, um nach dem Verzweigungsende wieder den gleichen Programmcode abzuarbeiten. Bei der Langform (benanntes if-then-else-Konstrukt) ist übrigens auch noch eine andere Variante möglich. So könnte der Bezeichner `name` nach `else` auch weggelassen werden. Nach dem `end if` ist er, sofern die benannte Form gewählt wurde, aber auf jeden Fall anzugeben.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  real      :: t, tc, tr
  character :: conv
  write( *, * ) "Gib Temperatur in Kelvin ein:"
  read( *, * ) t
  checkrange: if( t >= 0.0 ) then
    write( *, * ) "Umrechnung (c -> in Celsius, anderes Zeichen -> in
 Rankine):"
    read( *, * ) conv
```

```
    if( conv == "c") then
      tc = t - 273.15
      write( *, * ) tc
    else
      tr = 1.8 * t
      write( *, * ) tr
    end if
  else checkrange
    write( *, * ) "Fehler: t < 0.0 K"
  end if checkrange
end program bsp
```

24.2.4. else-if

Kurzform	Langform
if (logischer ausdruck 1) then if-anweisungsblock 1 else if (logischerAusdruck 2) then if-anweisungsblock 2 else if (logischerAusdruck n) then if-anweisungsblock n else else-anweisungsblock end if	name: if (logischer ausdruck 1) then if-anweisungsblock 1 else if (logischerAus-druck 2) then name if-anweisungsblock 2 else if (logischerAus-druck n) then name if-anweisungsblock n else name else-anweisungsblock end if name

Kurzform	Langform

Strukturbild

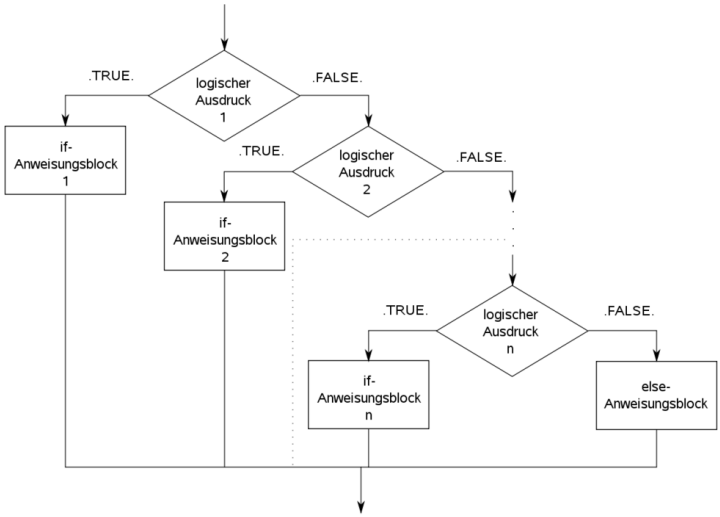


Abb. 27

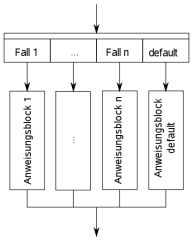
Das ist die allgemeinste Form der if-Verzweigung und wird im Fortran-Standard als "if construct" bezeichnet. Die vorher beschriebenen Formen sind nur vereinfachte Spezialfälle dieses Verzweigungstyps.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer :: i

  write( *, * ) "Gib eine natuerliche Zahl ein:"
  read( *, * ) i

! *** Langform ***
```

```
  i999: if( i == 1 ) then
    write( *, * ) "A"
  else if( i > 1 .and. i <= 5 ) then i999
    write( *, * ) "BCDE"
  else if( i > 5 .and. i <= 11 ) then i999
    write( *, * ) "FGHIJK"
  else i999
    write( *, * ) "L-Z"
  end if i999
! *** Das Gleiche in Kurzform ***
  if( i == 1 ) then
    write( *, * ) "A"
  else if( i > 1 .and. i <= 5 ) then
    write( *, * ) "BCDE"
  else if( i > 5 .and. i <= 11 ) then
    write( *, * ) "FGHIJK"
  else
    write( *, * ) "L-Z"
  end if
end program bsp
```

24.3. Die select case-Verzweigung

Kurzform	Langform	Strukturbild
select case(variable) case(fall1) anweisungsblock1 ! ... case(falln) anweisungsblockn [case default anweisungsblockdefault] end select	name: select case(variable) case(fall1) name anweisungsblock1 ! ... case(falln) name anweisungsblockn [case default name anweisungsblockdefault] end select name	 Abb. 28

Eine andere Möglichkeit zur Erstellung von Verzweigungen stellt die "select case"-Steueranweisung (das "case construct") bereit. Die `variable` kann vom Typ `integer`, `logical` oder `character` sein. Die Fälle werden durch Konstanten repräsentiert. Es muss nicht jeder Fall einzeln angeschrieben werden. Fälle, die zwar die gleichen Aktionen ausführen sollen, aber durch verschiedene Konstanten dargestellt werden, können zu einem Fall zusammengezogen werden. Optional kann auch noch ein "Default"-Fall angegeben werden.

Möglichkeiten um Fälle festzulegen:

n	Einzelne Konstante	gleich n
n1, n2, n3	Fallliste	n1 oder n2 oder n3
n:	Bereich	von n bis ...

:m	Bereich	von ... bis m
n:m	Bereich	von n bis m

Die in der Tabelle gelisteten Alternativen können natürlich auch kombiniert werden, z.B. eine Liste aus Einzelkonstanten und Bereichen. Bei logischem Datentyp ist die Angabe von Bereichen natürlich unsinnig.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer :: i

  read( *, * ) i
  select case( i )
    case( :111 )
      write( *, * ) 'Fall 1'
    case( 112:332, 334 )
      write( *, * ) 'Fall 2'
    case( 333 )
      write( *, * ) 'Fall 3'
    case default
      write( *, * ) 'unspezifiziertes Ereignis'
  end select
! Ausgabe (Eingabe: 222):
!    Fall 2
! Ausgabe (Eingabe: -5):
!    Fall 1
! Ausgabe (Eingabe: 333):
!    Fall 3
! Ausgabe (Eingabe: 5000):
!    unspezifiziertes Ereignis
end program bsp
```

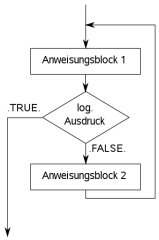
24.4. do-Schleifen

Fortran kennt nur einen allgemeinen Schleifentyp – die do-Schleife. Diese do-Schleife gibt es aber in verschiedenen Ausprägungen, so dass damit ziemlich alle denkbaren Einsatzfälle abgedeckt sind.

Im Prinzip haben alle do-Schleifen die Form

```
do [...] ... end do
```

24.4.1. do-if-exit

Kurzform	Langform	Strukturbild
do anweisungsblock1 if(logischer ausdruck) exit anweisungsblock2 end do	name: do anweisungsblock1 if(logischer ausdruck) exit anweisungsblock2 end do name	 Abb. 29

Lässt man den `anweisungsblock1` weg, dann erhält man eine kopfgesteuerte Schleife. Entfällt der `anweisungsblock2`, so ist die Schleife fußgesteuert. Ohne Abbruchbedingung ist dies eine Endlosschleife.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer :: i = 1
  do
    write( *, * ) i
    i = i + 1
    if( i > 10 ) exit
  end do
  write( * , * ) "Die do-Schleife wurde beendet"
! Ausgabe
!  1
!  2
!  ...
!  10
!  Die do-Schleife wurde beendet
end program bsp
```

24.4.2. Die do-Zählschleife

Kurzform	Langform
do zählvariable = startwert, endwert [, schrittweite] anweisungsblock end do	name: do zählvariable = startwert, endwert [, schrittweite] anweisungsblock end do name

Kurzform	Langform

Strukturbild

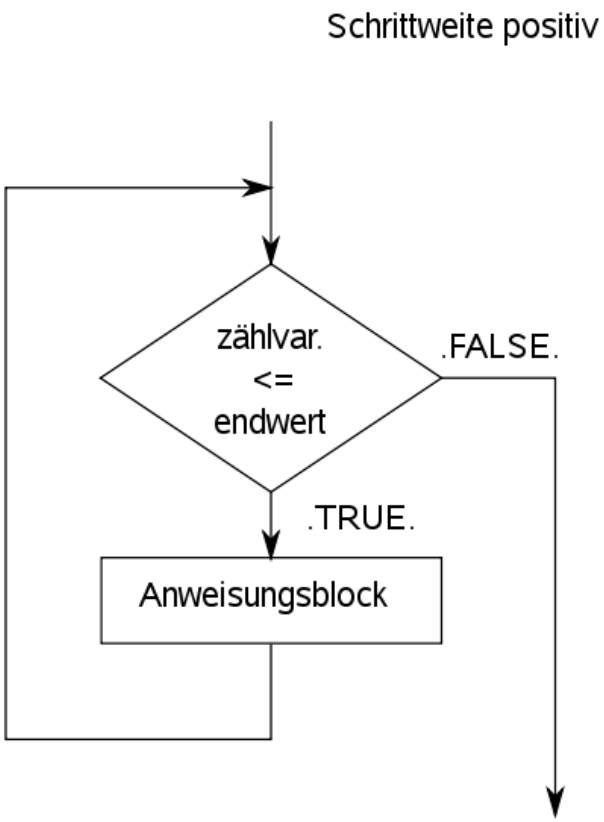


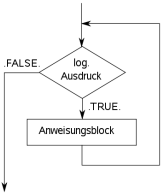
Abb. 30

Zählvariable, Startwert, Endwert und Schrittweite müssen vom Typ `integer` sein. Die Zählvariable darf in der Schleife nicht manipuliert werden. Wird die Schrittweite nicht explizit vorgegeben, so hat sie den Wert 1.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer :: i

  do i = 1, 10
    write( *, * ) i
  end do
! Zeilenweise Ausgabe der Zahlen 1 bis 10
end program bsp
```

24.4.3. do while

Kurzform	Langform	Strukturbild
do while(logischer ausdruck) anweisungsblock end do	name: do while(logischer ausdruck) anweisungsblock end do name	 Abb. 31

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer :: i

  i = 0

  do while( i < 5 )
    write( *, * ) i
    i = i + 1
  end do
!  Die Zahlen 0 bis 4 werden ausgegeben
end program bsp
```

24.4.4. Die implizite do-Liste

Bei Eingabe oder Ausgabe ist die Angabe einer impliziten do-Liste möglich.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer :: i

  write( *, * ) ( 'Hallo', i = 1, 10 )
! Ausgabe: HalloHalloHalloHalloHalloHalloHalloHalloHalloHallo
end program bsp
```

24.5. Spezialkonstrukte für Felder

24.5.1. where

Sollen in Feldern Elemente manipuliert werden, so können die `where`-Anweisungen (das "masked array assignment") hilfre-

ich sein. Dabei lassen sich Feldelementen in Abhängigkeit von vorgegebenen Bedingungen (Maske) neue Werte zuweisen.

Der where-Einzeiler

Form
where(bedingung) variable = ausdruck

Das "where statement" muss komplett in eine Zeile geschrieben werden, wobei Zeilenumbrüche mit dem &-Zeilenumbruchsmarker jedoch erlaubt sind. Für komplexere Probleme ist das im nächsten Abschnitt vorgestellte "where construct" (where-elsewhere) vorgesehen.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer, dimension(5) :: arr = (/ 5, 1, -1, 1, 7 /)

  write( *, * ) arr
! Ausgabe: 5  1  -1  1  7
  where( arr >= 3 ) arr = 99999
  write(*,*) arr
! Ausgabe: 99999  1  -1  1  99999
end program bsp
```

where-elsewhere

Kurzform	Langform
where(bedingung1) anweisungsblock [elsewhere(bedingung2) anweisungsblock2] [elsewhere(bedingungn) anweisungsblockn] [elsewhere anweisungsblockm] end where	name: where(bedingung1) anweisungsblock [elsewhere(bedingung2) name anweisungsblock2] [elsewhere(bedingungn) name anweisungsblockn] [elsewhere name anweisungsblockm] end where name

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer, dimension(10) :: arr =  (/ 5, 1, -1, 1, 7, -3, 7, 6, -9, 8
 /)

  where( arr <= 0 )
    arr = 0
  elsewhere( arr > 0 .and. arr <= 3 )
    arr = 1
  elsewhere
    arr = 2
  end where

  write(*,*) arr
! Ausgabe: 2  1  0  1  2  0  2  2  0  2
end program bsp
```

24.5.2. forall

Auch die `forall`-Schleife ist für den Einsatz bei Feldern gedacht. Im Gegensatz zum "where construct" werden hier die aus-

gewählten Feldelemente in erster Linie über die Indizes und erst in zweiter Linie über den Wert bestimmt.

Der forall-Einzeiler

Formen
forall(index = subscript:subscript[:stride]) anweisung
forall(ind1 = subs1a:subs1b[:str1], ind2 = subs2a:subs2b[:str2] [, indn = subsna:subsnb[:strn]]) anweisung

Die erste angegebene Form ist für eindimensionale Felder gedacht, die zweite für mehrdimensionale Felder. Über `subscript` kann der Indexbereich festgelegt werden. Das optionale `stride` gibt die Schrittweite vor. Wird dieser Wert nicht angegeben, so ist die Schrittweite = 1.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
  integer, dimension(5)    :: a = (/ 1, 2, 3, 4, 5 /)
  integer, dimension(2, 2) :: b = reshape( (/ 0, 2, -1, 5 /), (/ 2, 2
 /) )

  integer                  :: i, j
  forall( i = 1:3 ) a(i) = 0
  write( *, * ) a
! Ausgabe: 0  0  0  4  5
  forall( i = 1:5:2 ) a(i) = -1
  write( *, * ) a
! Ausgabe: -1  0  -1  4  -1
  forall( i = 1:5 ) a(i) = max ( a(i), 3 )
  write( *, * ) a
! Ausgabe: 3  3  3  4  3
  forall( i = 1:2, j = 2:2 ) b(i, j) = -9
  write( *, * ) b
! Ausgabe:  0  2  -9  -9
```

```
end program bsp
```

Aber das ist nicht alles, was der forall-Einzeiler zustande bringt. Zusätzlich zur Feldelementbestimmung über Indexbereiche kann auch eine Maske vorgegeben werden. Die forall-Schleife ist also auch eine erweiterte where-Anweisung.

Form

forall(ind1 = s1a:s1b[:s1] [, indn = sna:snb[:sn] [, maske])
anweisung

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
  integer, dimension(5)    :: a = (/ 1, 2, 3, 4, 5 /)

  integer                  :: i, j
  forall( i = 1:4, a(i) > 2 ) a(i) = 0
  write( *, * ) a
! Ausgabe: 1  2  0  0  5
end program bsp
```

Bei der Verwendung von forall-Anweisungen sind einige Prämissen zu beachten. So müssen nebst anderem in Masken oder im Anweisungsbereich verwendete Funktionen `pure` sein. Was das genau bedeutet wird später im Kapitel UNTERPROGRAMME[1] erläutert

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
  integer, dimension(5)    :: a = (/ 1, 2, 3, 4, 5 /)
```

1 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20FORTRAN%2095%3A%20UNTERPROGRAMME%23DIE%20PURE%20ANWEISUNG

```
  integer                  :: i
! Die Funktion berechne_1() darf nicht innerhalb einer
 forall-Anweisung
! aufgerufen werden, da sie nicht "pure" ist
!  forall( i = 1:4, a(i) > 2 ) a(i) = berechne_1( a(i) ) !
 FEHLERMELDUNG

! Die Funktion berechne_2 macht das Gleiche wie berechne_1(),
! ist aber als "pure" gekennzeichnet und darf in der forall-Anweisung
! aufgerufen werden
  forall( i = 1:4, a(i) > 2 ) a(i) = berechne_2( a(i) ) ! OK

  write( *, * ) a
! Ausgabe: 1  2  9  12  5
  contains
    integer function berechne_1( val )
      integer, intent( in ) ::  val
      integer               ::  res

      res = val * 3
      berechne_1 = res
    end function berechne_1

    pure integer function berechne_2( val )
      integer, intent( in ) ::  val
      integer               ::  res

      res = val * 3
      berechne_2 = res
    end function berechne_2
end program bsp
```

Das "forall construct"

Kurzform	forall(index = subscript:subscript[:stride]) anweisungsblock end forall
Langform	name: forall(index = subscript:subscript[:stride]) anweisungsblock end forall name

Es gilt im Prinzip das gleiche wie für den forall-Einzeiler. Es sind innerhalb der Schleife eben mehrere Anweisungen erlaubt. Es gelten auch die Formen für mehrdimensionale Felder und mit Vorgabe einer Maske. Diese sind in diesem Abschnitt aber nicht mehr explizit angeführt.

24.6. Terminierung

24.6.1. stop

Die `stop`-Anweisung beendet das Programm.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
  write( *, * ) 'Vor Stop-Statement'
  stop
  write( *, * ) 'Nach Stop-Statement'
! Ausgabe:
!   Vor Stop-Statement
end program bsp
```

Der stop-Anweisung kann auch ein "stop-code" mitgegeben werden. Dieser "stop-code" kann eine Ganzzahl (maximal 5-stellig) oder eine Zeichenkette sein. Der "stop-code" wird bei der Programmtermination ausgegeben. Wo die Ausgabe erfolgt und wie sie aussieht ist aber betriebssystem- und compilerabhängig.

Fortran 90/95-Code (free source form)

```
program bsp
  implicit none

  integer :: i

  read( *, * ) i
  if( i == 1 ) then
    stop 999
```

```
  else
    stop "Sonstiger Stop"
  end if
! Ausgabe (bei Eingabe von 1):
! ifort (Linux):
!   999
! gfortran, g95 (Linux):
!   STOP 999
! Sun f95 (Linux):
!   STOP: 999
end program bsp
```

24.6.2. exit

Mit der Anweisung `exit` kann eine do-Schleife verlassen werden. Wird eine do-Schleife mit einer Bezeichnung versehen, so kann bei der `exit`-Anweisung explizit auf die benannte Schleife Bezug genommen werden. Wird kein Schleifenname angegeben, so bezieht sich die `exit`-Anweisung immer auf die innerste Schleife, mit der sie im Zusammenhang steht.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
  integer :: i, j

  outerloop: do i = 1, 5
    write( *, * ) "Outer", i, "Beginn"

    innerloop:  do j = 1, 3
      if( i == 1 )  exit
      if( i == 2 )  exit innerloop
      if( i == 4 )  exit outerloop
      write( *, * ) "Inner", j
    end do innerloop

    write( *, * ) "Outer", i, "Ende"
  end do outerloop
! Ausgabe:
!  Outer 1 Beginn
!  Outer 1 Ende
```

```
!  Outer 2 Beginn
!  Outer 2 Ende
!  Outer 3 Beginn
!  Inner 1
!  Inner 2
!  Inner 3
!  Outer 3 Ende
!  Outer 4 Beginn
end program bsp
```

24.6.3. cycle

Mit `cycle` wird der aktuelle do-Schleifendurchlauf beendet und wieder zum Schleifenkopf gesprungen. Wird eine do-Schleife mit einer Bezeichnung versehen, so kann bei der `cycle`-Anweisung explizit auf die benannte Schleife Bezug genommen werden. Wird kein Schleifenname angegeben, so bezieht sich die `cycle`-Anweisung immer auf die innerste Schleife, mit der sie im Zusammenhang steht.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
  integer :: i, j

  outerloop: do i = 1, 5
    write( *, * ) "Outer", i, "Beginn"

    innerloop:  do j = 1, 3
      if( i == 1 )  cycle outerloop
      if( i == 4 )  cycle innerloop
      if( i == 5 )  cycle
      write( *, * ) "Inner", j
    end do innerloop

    write( *, * ) "Outer", i, "Ende"
  end do outerloop
! Ausgabe:
!   Outer           1 Beginn
!   Outer           2 Beginn
!   Inner           1
```

```
!   Inner           2
!   Inner           3
!   Outer           2 Ende
!   Outer           3 Beginn
!   Inner           1
!   Inner           2
!   Inner           3
!   Outer           3 Ende
!   Outer           4 Beginn
!   Outer           4 Ende
!   Outer           5 Beginn
!   Outer           5 Ende
end program bsp
```

25. Datentypen höherer Genauigkeit

Fortran 95 bietet für die Festlegung von Datentypen höherer Genauigkeit einen neuen Ansatz.

25.1. Einfache Variante

Mittels `kind` (dt.: Art, Sorte, Gattung, Wesen) kann die Genauigkeit bzw. der Wertebereich eines Datentyps festgelegt werden. Die einfache Variante ist aber system- und compilerabhängig.

datentyp (kind=*wert*) :: *var*

Kurzform:

datentyp (*wert*) :: *var*

Für `wert` ist meist (aber nicht immer) die Anzahl der Bytes einzusetzen. Literale eines solchen Datentyps sind mit einem entsprechenden Anhängsel zu kennzeichnen, z.B.:

- 3.141592654_8
- -4.9e55_8

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  real(kind=8) :: variable
  ! variable ist nun vom Datentyp "double precision"

  variable = 1.55555555555_8

  write(*,*) variable
  ! Ausgabe: 1.55555555555000

  ! Hier wird nur eine gewöhnliche real-Zahl mit 7 Nachkommastellen
  ! zugewiesen
  variable = 1.55555555555

  write(*,*) variable
  ! Ausgabe: 1.55555558204651
end program bsp
```

25.2. System- und compilerunabhängige Variante

Das `kind`-Attribut darf über eine Konstante oder eine benannte Konstante belegt werden. So kann statt

```
real(kind=8) :: var
```

auch

```
integer, parameter :: dp = 8
real(kind=dp)      :: var
```

geschrieben werden. Der entsprechende `kind`-Wert ist auch per Funktion ermittelbar. Die Variable `var`

```
integer, parameter :: dp = kind(0.0d0)
real(kind=dp)      :: var
```

ist in diesem Beispiel somit vom Datentyp `double precision`. Das ist schon compilerunabhängig verwendbarer Fortran-Code.

Nachfolgend wird die `selected_real_kind(p, r)`-Funktion vorgestellt, die durch Vorgabe der gewünschten Nachkommastellen und/oder des Exponentenmaximalwertes einen potentiell geeigneten `kind`-Wert für Gleitkommazahlen ermittelt.

25.2.1. selected_real_kind

integer, parameter :: *name* = selected_real_kind(*anzahl_nachkommastellen, max_exponentenwert*)

Die Funktion `selected_real_kind` gibt einen Wert zurück, der alle Gleitkommazahlen mit mindestens `anzahl_nachkommastellen` Dezimalstellen Genauigkeit und einem Exponentenbereich von `max_exponentenwert` berücksichtigt. Gibt es keinen solchen systemspezifischen Wert, so wird einer der folgenden Werte zurückgegeben:

- -1 ... die geforderte Anzahl an Dezimalstellen ist nicht verfügbar
- -2 ... der Exponentenbereich ist nicht verfügbar
- -3 ... nichts von beidem ist verfügbar

Wird eine solche Konstante mit negativem Wert nachfolgend als kind-Wert bei der Deklaration einer Variablen, Konstanten etc. verwendet, so führt dies in aller Regel zu einer Fehlermeldung bereits beim Kompiliervorgang.

Es ist auch möglich die `selected_real_kind`-Funktion mit nur einem Argument aufzurufen. Die Argumente sind mit `p` (für *precision*) bzw. `r` (für *range*) benannt.

Beispiele:

```
integer, parameter :: dp = selected_real_kind(r=60)
```

```
integer, parameter :: dp = selected_real_kind(p=15)
```

```
integer, parameter :: ultrap = selected_real_kind(1000, 5000) !
liefert höchstwahrscheinlich
                                                              ! die
negative Zahl -3
```

25.2.2. Variablendeklaration

real (kind=*name*) :: variable

Kurzform:

real (*name*) :: variable

Beispiel:

```
integer, parameter :: dp = selected_real_kind(r=60)
real(kind=dp)      :: var
```

25.2.3. Konstanten

zahl_*name*

Literale eines solchen Datentyps sind mit dem im Programmcode festgelegten charakteristischen Anhängsel zu kennzeichnen, z.B.:

- 3.141592654_dp
- -4.9e55_dp

25.2.4. Beispiel

Fortran 90/95-Code (free source form)

```
program bsp
  implicit none

  integer, parameter :: dp = selected_real_kind(15, 300)

  real(kind=dp) :: variable

  variable = 1.5555555555_dp / 2_dp

  write(*,*) variable
  ! Ausgabe: 0.777777777750000
  write(*,*) kind(variable)
  ! Ausgabe: 8
end program bsp
```

26. Datenverbund

26.1. Einleitung

Datenelemente lassen sich auch zu eigenen zusammengesetzten Datentypen verbinden. Der Datenverbund ist in anderen Programmiersprachen auch als Struktur (structure, struct) bekannt. Der Fortran 95-Standard spricht vom "derived type", also einem abgeleiteten Datentyp. Zulässig als Komponenten eines Datenverbundes sind Skalare, Felder, Zeiger oder auch andere Datenverbunde. Ein Datenverbund ist durch das Schlüsselwort `type` gekennzeichnet und kann in folgenden Formen auftreten

type name ... end type [name]	type [,zugriffsspezifizierer] :: name ... end type [name]

Die Elemente in eckigen Klammern sind optional. Die Angabe eines Zugriffsspezifizierers (`public`, `private`) ist nur bei Verwendung von Datenverbunden in Modulen erlaubt.

26.2. Einen Datenverbund anlegen

type :: name [sequence] ! optional sequentielle Speicherplatz-ablage datentyp :: komponentenname_1 ! ... datentyp :: komponentenname_n end type [name]

Beispiel (Programmausschnitt): **Fortran 90/95-Code (free source form)**

```
! ...
  type :: koord
    integer            :: id
    real, dimension(3) :: x
  end type koord

! ...
```

26.3. Eine Variable vom Typ "Datenverbund" anlegen

Die Variablendeklaration erfolgt in der Form

type(name) :: var

Beispiel (Programmausschnitt): **Fortran 90/95-Code (free source form)**

```
! ...
  type :: koord
    integer            :: id
    real, dimension(3) :: x
  end type koord

  type( koord ) :: k
! ...
```

26.4. Zugriff auf Einzelkomponenten eines Datenverbunds

Eine Einzelkomponente in einem Datenverbund lässt sich als Kombination des Datenverbundvariablenbezeichners und des Komponentennamen ansprechen. Verknüpft werden diese beiden Bezeichner mit dem %-Zeichen.

```
var%komponentenname
```

Beispiel (Programmausschnitt): **Fortran 90/95-Code (free source form)**

```
! ...
  type :: koord
    integer            :: id
    real, dimension(3) :: x
  end type koord

  type( koord ) :: k
  k%id = 999
  k%x(1) = 0.0
  k%x(2) = 20.5
  k%x(3) = 10.0
  write( *, * ) k%x(2)
! ...
```

26.5. Zugriff auf einen Datenverbund als Ganzes

Ein Datenverbund kann in bestimmten Situation nicht nur komponentenweise angesprochen werden, sondern auch als Einheit, z.B. bei der Ein-/Ausgabe oder beim Zuweisen der Werte einer Datenverbundvariablen an eine andere.

Beispiel (Programmausschnitt): **Fortran 90/95-Code (free source form)**

```
! ...
  type :: koord
    integer            :: id
    real, dimension(3) :: x
  end type koord

  type( koord ) :: k1, k2
  read( *, * ) k1
  k2 = k1
  write( *, * ) k2
! ...
```

26.6. Elemente eines Datenverbunds initialisieren

26.6.1. Default-Werte im Datenverbund setzen

Innerhalb eines Datenverbundes lassen sich die einzelnen Variablen mit Vorgabewerten belegen, z.B.

```
integer                       :: i = 5
character, dimension( 2, 2 ) :: c = reshape( (/ "A", "B", "C", "D"
/), (/ 2, 2 /) )
```

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  type :: koord
    integer              :: id = -999999
    real, dimension( 3 ) :: x  = 0.0
  end type koord

  type( koord ) :: k

  write( *, * ) k

! Ausgabe:
```

```
!   -999999   0.0   0.0   0.0
end program bsp
```

26.6.2. Der "structure constructor"

Die Initialisierung der Datenverbundelemente bzw. eine spätere Wertzuweisung kann auch per "structure constructor" ("derived type value constructor") vorgenommen werden.

name(wertliste)

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  type :: koord
    integer              :: id
    real, dimension( 3 ) :: x
  end type koord

  type( koord ) :: k = koord( 12, (/ 0.0, 1.5, 20.5 /) )
  write( *, * ) k
  k = koord( 13, (/ 5.5, 0.0, 0.5 /) )
  write( *, * ) k

! Ausgabe:
!   12   0.0   1.5   20.5
!   13   5.5  0.0   0.5
end program bsp
```

Enthält ein Datenverbund bereits Variablen mit Vorgabewerten, so werden diese durch die Werte im "structure constructor" überschrieben.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
```

```
  type :: person
    character(25) :: vorname = 'NN'
    character(25) :: nachname = 'NN'
    integer       :: alter
  end type person

  type(person), dimension(5) :: p

  p(1) = person('Harri', 'P.', 41)
  p(5) = person('Creszenzia', 'T.', 18)

  write(*, fmt=111) p
! Ausgabe:
!   Harri                    P.                        41
!   NN                       NN                         0
!   NN                       NN                         0
!   NN                       NN                         0
!   Creszenzia               T.                        18
  write(*,*) p(1)%vorname
! Ausgabe:
!   Harri

  p(2)%vorname = 'Holger'
  write(*, '(A)') p%vorname
! Ausgabe:
!   Harri
!   Holger
!   NN
!   NN
!   Creszenzia

 111 format(2A, I3)
end program bsp
```

Beispiel: Verwendung eines Datenverbunds in einem anderen Datenverbund **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
  type :: tripel
    real :: x, y, z
  end type tripel

  type :: koerper
    integer        :: id
    type( tripel ) :: bezugskoordinate
    type( tripel ) :: orientierung
```

```
  end type koerper

  type( koerper ) :: k

  k = koerper( 1005, tripel( 10.5, 0.0, -6.5 ), tripel( 0.0, 0.0, -0.8
 ) )

  write( *, * ) k
! Ausgabe:
!   1005   10.50000       0.000000      -6.500000       0.000000

!   0.000000     -0.8000000
end program bsp
```

26.7. Das Attribut sequence

Im Datenverbund kann zu Beginn auch das Attribut sequence angegeben werden. Dieses spezifiziert, dass die Werte eines Datenverbunds sequentiell gespeichert werden. Wird dieses Attribut weggelassen, so ist nicht gewährleistet, dass die Werte von Datenverbundvariablen in der selben Reihenfolge, wie sie im Datenverbund gereiht wurden im Speicher wiederzufinden sind, oder dass sie zusammenhängend gespeichert werden. Oft wird die Speicherreihenfolge keine wesentliche Rolle spielen. Ein Spezialfall, wo die Speicherreihenfolge jedoch essenziell ist, wird anschließend dargestellt.

26.8. Die Verwendung eines Datenverbunds in separaten Programmeinheiten

Ein Datenverbund kann über ein Modul verschiedenen Programmeinheiten bekannt gemacht werden. Dazu aber später. Es ist nämlich auch so möglich in komplett getrennten Programmein-

heiten auf ein und denselben Datenverbund-Datentyp zuzugreifen.

Beispiel: Ein ähnliches Beispiel findet sich auch im J3/97-007R2 Working Draft, Note 4.31 **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  type :: koord
    sequence
    integer              :: id
    real, dimension( 3 ) :: x
  end type koord

  type( koord ) :: k = koord( 555, (/ 3.0, 4.0, 5.5 /) )

  call ausgabe( k )

! Ausgabe
!    555   3.0000000   4.0000000   5.5000000
end program bsp
subroutine ausgabe( coord )
  type :: koord
    sequence
    integer              :: id
    real, dimension( 3 ) :: x
  end type koord

  type( koord ), intent( in ) :: coord

  write( *, * ) coord
end subroutine ausgabe
```

In diesem speziellen Fall muss der Datenverbund lt. Standard mit dem Attribut `sequence` versehen werden. Desweiteren muss der im Hauptprogramm deklarierte Datenverbund komplett identisch mit jenem im Unterprogramm sein (Datenverbundname, Variablenbezeichner, Reihenfolge, Datentypen). In der Praxis sehen das viele Compiler nicht so eng. Allerdings können bei Nichtbeachtung dieser Vorgaben spätestens beim Wechsel des Compilers oder auf andere Rechnerarchitekturen gröbere Probleme auftreten.

27. Standardfunktionen

27.1. Tabellenlegende

Abkürzung	Beschreibung
i	Integer-Datentyp (integer)
r	Real-Datentyp (real)
x	Complex-Datentyp (complex)
d	Double-precision-Datentyp (real(z, kind(0.0D0)))
z	beliebiger numerischer Datentyp (integer, real, complex)
c	Zeichen (character)
l	Logical-Datentyp (logical)
any	beliebiger intrinsischer Datentyp
arr	Feld (Array)
aarr	dynamisches Feld
ptr	Zeiger (Pointer)

Die in den nachfolgenden Unterkapiteln angeführten Tabellen und Listen geben die im Fortran 95-Working Draft gelisteten intrinsischen Funktionen in simplifizierter Form wieder. Auch die Standard-Subroutinen wurden berücksichtigt. Die einzelnen Compiler kennen zum Teil wesentlich mehr Funktionen als im

Standard vorgegeben. Bei Verwendung solcher Funktionen sind die Programme jedoch nicht mehr quellcodekompatibel. Auf die Wiedergabe solcher compilerspezifischen Funktionen wird hier deshalb verzichtet.

27.2. Datentypfunktionen[1]

- Umwandlung in INTEGER
- Umwandlung in REAL
- Umwandlung in DOUBLE PRECISION
- Umwandlung in COMPLEX
- Umwandlung in CHARACTER
- kind-Parameter

27.3. Mathematische Funktionen[2]

- Rundung
- Absolutwert
- Modulo
- Vorzeichentransfer
- Positive Differenz
- Maximum
- Minimum
- Komplexe Zahlen
- Quadratwurzel
- Exponentialfunktion

1 http://de.wikibooks.org/wiki/Fortran%3A%20Fortran%2095%3A%20Standardfunktionen%3A%20Datentypfunktionen%20

2 http://de.wikibooks.org/wiki/Fortran%3A%20Fortran%2095%3A%20Standardfunktionen%3A%20Mathematische%20Funktionen%20

- Logarithmen
- Winkelfunktionen
- Arkusfunktionen
- Hyperbelfunktionen

27.4. STRINGFUNKTIONEN[3]

- Lexikalische Funktionen
- Sonstige

27.5. FELDFUNKTIONEN[4]

- Konstruktion und Umgestaltung von Feldern
- Abfragen von Feldstatus, Felddaten und Feldmetadaten
- Funktionen für Vektoren und Matrizen
- Sonstige

3 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20FORTRAN%2095%3A%20STANDARDFUNKTIONEN%3A%20STRINGFUNKTIONEN%20

4 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20FORTRAN%2095%3A%20STANDARDFUNKTIONEN%3A%20FELDFUNKTIONEN%20

27.6. Zeigerfunktionen[5]

27.7. Bitfunktionen[6]

27.8. Weitere Funktionen[7]

27.9. Intrinsische Subroutinen[8]

5 http://de.wikibooks.org/wiki/Fortran%3A%20Fortran%2095%3A%20Standardfunktionen%3A%20Zeigerfunktionen%20

6 http://de.wikibooks.org/wiki/Fortran%3A%20Fortran%2095%3A%20Standardfunktionen%3A%20Bitfunktionen%20

7 http://de.wikibooks.org/wiki/Fortran%3A%20Fortran%2095%3A%20Standardfunktionen%3A%20Weitere%20Funktionen%20

8 http://de.wikibooks.org/wiki/Fortran%3A%20Fortran%2095%3A%20Standardfunktionen%3A%20Intrinsische%20Subroutinen%20

28. Unterprogramme

Werden Programme umfangreicher und komplexer oder werden einzelne Programmabschnitte öfter verwendet, dann ist es sinnvoll Unterprogramme (Prozeduren) zu verwenden. Fortran 95 kennt zwei Typen von Unterprogrammen:

- Funktionen (`function`)
- Subroutinen (`subroutine`)

Darüber hinaus existiert in Fortran die Möglichkeit Unterprogramme und Daten mittels Modulen in das Programm einzubeziehen.

28.1. Das `function` Unterprogramm

Eine Funktion ist ein Unterprogramm, das genau einen Wert zurück gibt. Welcher Art von Datentyp der Wert ist, wird durch eine Zuweisung an den Funktionsnamen erreicht (z. B. `real`).

Die allgemeine Funktionsdeklaration lautet:

datentyp function funktionsname ([formale parameter]) Vereinbarungsteil Anweisungsteil funktionsname = wert [return] end function funktionsname

Wird der Dateityp bei der Funktionsdeklaration nicht angegeben, so muss der Datentyp der Funktion im Vereinbarungsteil festgelegt werden:

function funktionsname ([formale parameter]) Vereinbarungsteil datentyp :: funktionsname Anweisungsteil funktionsname = wert [return] end function funktionsname

Mittels `return` kann zur aufrufenden Programmeinheit zurückgekehrt werden. Ein `return` unmittelbar vor der `end`-Anweisung ist nicht unbedingt erforderlich, da das Programm beim Erreichen der `end`-Anweisung zur aufrufenden Programmeinheit zurück springt. Deswegen sollte bei allen neueren Programmen darauf verzichtet werden. Wichtig ist die `return`-Anweisung für alternative Rücksprünge zur aufrufenden Programmeinheit.

Eine Funktion wird meist mittels folgender allgemeiner Anweisung von der ausführenden Programmeinheit aufgerufen:

variable = funktionsname(aktuelle parameter)

Dabei orientiert sich diese Unterprogrammtechnik direkt an mathematischen Funktionen. Im folgenden Beispiel ruft das Programm `test` die Funktion `funk` auf, führt das Unterprogramm aus und gibt den entsprechenden Rückgabewert an das Programm `test` zurück:

Datei *bsp.f95*: **Fortran 90/95-Code (free source form)**

```
program test
  !
  implicit none
  real :: funk
  real :: x, y
  write(*,*) 'x -> '
  read(*,*) x
  y = funk( x )
```

```
  write(*,*) 'funk -> ', y
end program test
real function funk( a )
  !
  implicit none
  real :: a
  funk = a**2
end function funk
```

Das Unterprogramm `funk` kann sich direkt unter der aufrufenden Programmeinheit in der *selben* Textdatei oder in einer *separaten* Datei befinden. Befindet sich das Unterprogramm in einer separaten Datei, so muss diese und die aufrufende Programmeinheit zusammen kompiliert werden:

Übersetzung mit *gfortran* im vorliegenden Fall:

```
gfortran -o bsp bsp.f95
```

Übersetzung mit *gfortran* bei separaten Dateien (bspw.):

```
gfortran -o bsp bsp.f95 funk.f95
```

Es ist aber auch möglich eine Funktion nur durch ihren Namen aufzurufen. Der Rückgabewert der Funktion wird dann direkt ausgegeben:

Datei *bsp.f95*: **Fortran 90/95-Code (free source form)**

```
program test
  !
  implicit none
  real :: funk
  real :: x
  write(*,*) 'x -> '
  read(*,*) x
  write(*,*) 'funk -> ', funk( x )
end program test
real function funk( a )
```

```
  !
  implicit none
  real :: a
  funk = a**2
end function funk
```

28.2. Das `subroutine` Unterprogramm

Eine `subroutine` ist ein Unterprogramm, die mehr als einen Wert zurückgeben kann. Eine Subroutine besitzt im Gegensatz zu einer Funktion keinen Datentyp und Rückgabewert.

Die allgemeine Deklaration einer Subroutine lautet:

subroutine subroutinenname ([formale parameter]) Vereinbarungsteil Anweisungsteil [return] end subroutine subroutinenname

Mittels `return` kann zur aufrufenden Programmeinheit zurückgekehrt werden. Ein `return` unmittelbar vor der `end`-Anweisung ist nicht unbedingt erforderlich, da das Programm beim Erreichen der `end` zur aufrufendenen Programmeinheit zurück springt. Deswegen sollte bei allen neueren Programmen darauf verzichtet werden. Wichtig ist die `return` Anweisung für alternative Rücksprünge zur aufrufenden Programmeinheit.

Aufgerufen wird eine Subroutine aus der aufrufenden Programmeinheit mittels der `call`-Anweisung:

call *subroutinenname* [([*aktuelle parameter*])]

Im folgenden Beispiel wird die Subroutine `sub` aufgerufen. Diese gibt dann zwei Werte zurück:

Datei *bsp.f95* **Fortran 90/95-Code (free source form)**

```
program bsp
  !
  implicit none
  real :: x, y, z
  write( *, * ) 'x -> '
  read( *, * ) x
  call sub( x, y, z )
  write( *, * ) 'y -> ', y, 'z -> ', z
end
subroutine sub( a, b, c )
  !
  implicit none
  real :: a, b, c
  b = a**2
  c = a**2 + b
end subroutine sub
```

28.3. Weitere Anweisungen für die Unterprogrammtechnik

28.3.1. Das `intent`-Attribut

Im Vereinbarungsteil der Funktion wird der Parameterdeklaration das Schlüsselwort `intent` mitgegeben.

```
datentyp, intent( in ) :: var
```

Mit `intent( in )` wird angezeigt, dass der Parameter `var` in der Funktion nicht geändert wird und kein Rückfluss der Information in die aufrufende Programmeinheit stattfindet. Ein `intent( out )` oder `intent( inout )` wie bei Subroutinen wäre meist auch möglich, widerspricht aber dem Grundgedanken des Fortran-Funktionsbegriffs und der strukurierten Programmierung. Bei

einer Funktion soll der Informationsrückfluss über den Rückgabewert stattfinden und nicht über Parameter.

Bei der Parameterübergabe bietet eine Subroutine folgende `intent`-Möglichkeiten:

- *datentyp*, intent(in) :: *var* ... Informationfluss von der aufrufenden Programmeinheit in die Funktion
- *datentyp*, intent(out) :: *var* ... Informationfluss von der Subroutine zur aufrufenden Programmeinheit
- *datentyp*, intent(inout) :: *var* ... beidseitiger Informationsfluss

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
  real :: a, b, c

  a = 1.0
  b = 2.0
  c = 3.0

  call sub(a, b, c)

  write (*,*) 'Hauptprogramm: ', a, b, c
  ! Ausgabe: 1.000000       22.20000       33.30000
end
subroutine sub(x, y, z)
  implicit none
  real, intent(in)    :: x
  real, intent(out)   :: y
  real, intent(inout) :: z

  write (*,*) 'Unterprogramm: ', x, y, z
  ! Ausgabe: 1.000000       2.000000       3.000000

  y = 22.2
  z = 33.3

end subroutine sub
```

Die aktuellen und formalen Parameter müssen hinsichtlich Datentyp, Anzahl, Reihenfolge übereinstimmen.

28.3.2. Die `pure` Anweisung

Ein Unterprogramm welches keine Seiteneffekte hat ist eine bloßes bzw. reines (pure) Unterprogramm. Ein Unterprogramm erzeugt dann keine Seiteneffekte, wenn es weder seine Eingabedaten, noch die Daten verändert, die außerhalb des Unterprogrammes liegen, es sei denn, es wären seine Ausgabedaten. In einem reinen Unterprogramm haben die lokalen Variablen keine `save` Attribute, noch werden die lokalen Variablen in der Datendeklaration initialisiert.

Reine Unterprogramme sind für das `forall`-Konstrukt notwendig: das `forall`-Konstrukt wurde für das parallele Rechnen konzipiert, weshalb hier der Computer entscheidet, wie das Konstrukt abgearbeitet werden soll. Dazu ist es aber notwendig, das es egal ist in welcher Reihenfolge das Konstrukt abgearbeitet wird. Gilt dies nicht - hat das Unterprogramm also Seiteneffekte - so kann das `forall`-Konstrukt nicht verwendet werden.

Jedes Ein- und Ausgabeargument in einem reinen Unterprogramm muss mittels des `intent`-Attributs deklariert werden. Darüberhinaus muss jedes Unterprogramm, das von einem reinen Unterprogramm aufgerufen werden soll, ebenfalls ein reines Unterprogramm sein. Sonst ist das aufrufende Unterprogramm kein reines Unterprogramm mehr.

28.3.3. Die `elemental` Anweisung

Ein Unterprogramm ist elementar, wenn es als Eingabewerte sowohl Skalare als auch Felder akzeptiert. Ist der Eingabewert ein Skalar, so liefert ein elementares Unterprogramm einen Skalar als Ausgabewert. Ist der Eingabewert ein Feld, so ist der Ausgabewert ebenfalls ein Feld.

28.3.4. Die `return` Anweisung

Eine `return`-Anweisung darf nur im Gültigkeitsbereich von Unterprogrammen verwendet werden. Sie bewirkt einen Abbruch der Unterprogrammausführung und eine Rückkehr zum Aufrufpunkt, wo mit der nächsten Anweisung fortgesetzt wird.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program main
  implicit none

  call xyz( -2 )
  write( *, * ) "UP-Ende"
  stop
  write( *, * ) "Programmende"
  contains
    subroutine xyz( n )
      integer, intent( in ) :: n
      integer               :: k

      do k = n, n+20
        write( *, * ) k
    if( k == 5 ) return
      end do
      write( *, * ) "k_max =", k
    end subroutine xyz

! Ausgabe:
!          -2
!          -1
!           0
!           1
!           2
!           3
!           4
!           5
!   UP-Ende
end program main
```

Einige Compiler erlauben zwecks Programmabbruch auch ein `return` anstelle des `stop` im Hauptprogramm. Das ist aber nicht standardkonform. Andere Compiler würden solchen Code mit

einer Fehlermeldung abweisen, das Programm wäre somit nicht mehr uneingeschränkt portabel.

Ein `exit` in der Schleife anstelle der `return`-Anweisung würde nur den Schleifendurchlauf abbrechen und die Unterprogrammausführung würde nach der Schleife fortgesetzt.

28.4. Felder als Parameter

Beispiel: Übergabe eines ganzen Feldes

Datei *bsp.f95*: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer, dimension(3,3) :: feld
  integer :: cnt, i, j

  cnt = 1

  do i = 1, 3
    do j = 1, 3
      feld(j,i) = cnt
      cnt = 1 + cnt
    end do
  end do

  !  Unterprogrammaufruf
  call sub(feld)
  !  Ausgabe: 1   2   3   4   5   6   7   8   9
end program bsp
```

Datei *sub.f95* **Fortran 90/95-Code (free source form)**

```
subroutine sub(arr)
  implicit none
  integer, dimension(3,3), intent(in) :: arr

  write(*,*) arr
```

```
end subroutine sub
```

Beispiel: Übergabe einer Feld-Teilkette

Datei *bsp.f95*: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer,dimension(3,3) :: feld
  integer :: cnt, i, j

  cnt = 1

  do i = 1, 3
    do j = 1, 3
      feld(j,i) = cnt
      cnt = 1 + cnt
    end do
  end do

  ! Unterprogrammaufruf
  call sub(feld(1:2,2:3))
  !  Ausgabe: 4    5    7     8
end program bsp
```

Datei *sub.f95* **Fortran 90/95-Code (free source form)**

```
subroutine sub(arr)
  implicit none
  integer, dimension(0:1, 0:1), intent(in) :: arr

  write(*,*) arr

end subroutine sub
```

Beispiel: Übergabe eines Feld-Einzelelements

Datei *bsp.f95*: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer, dimension(3,3) :: feld
  integer cnt, i, j
```

```
  cnt = 1

  do i = 1, 3
    do j = 1, 3
      feld(j,i) = cnt
      cnt = 1 + cnt
    end do
  end do

  ! Unterprogrammaufruf
  call sub(feld(1,2))
  ! Ausgabe: 4
end program bsp
```

Datei *sub.f95* **Fortran 90/95-Code (free source form)**

```
subroutine sub(arr)
  implicit none
  integer, intent(in) :: arr

  write(*,*) arr
end subroutine sub
```

28.5. Prozeduren als Parameter

Auch Prozeduren können als Parameter übergeben werden.

Standardfunktionen (intrinsic functions) werden dazu folgendermaßen im Vereinbarungsteil gekennzeichnet:

intrinsic *namensliste*

oder

datentyp, intrinsic :: *namensliste*

Eigene Unterprogramme oder Unterprogramme aus Bibliotheken mit:

external *namensliste*

oder

datentyp, external :: *namensliste*

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  real, parameter :: PI=3.1415927

! intrinsic functions
  intrinsic sin, cos

! Unterprogrammaufrufe
  call sub(sin, PI)
! Ausgabe: 0.000000
  call sub(cos, PI)
! Ausgabe: -1.000000
end program bsp

subroutine sub(funk, x)
  implicit none
  real :: funk, x
  write(*,*) nint(funk(x)*1000)/1000.0
end subroutine sub
```

28.6. Optionale Parameter

Ab Fortran 90 sind auch optionale Parameter für Unterprogramme erlaubt. Solche Parameter sind durch das Attribut `optional` zu kennzeichnen. Auch der aufrufenden Programmeinheit muss diese Parameter-Optionalität bekannt gegegeben werden, z.B. über ein Interface. Das aktuelle Vorhandensein eines als optional

markierten Parameters beim Unterprogrammaufruf kann im Unterprogramm selbst durch die Funktion `present` geprüft werden.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  interface
    subroutine ab(a, b)
      integer, intent( in )           :: a
      integer, intent( in ), optional :: b
    end subroutine ab
  end interface

  call ab( 1 )
  call ab( 8, 12 )
! Ausgabe:
!   Nur a gegeben  1
!   Beide Parameter gegeben  8 12
end program bsp
subroutine ab(a, b)
  integer, intent( in )           :: a
  integer, intent( in ), optional :: b

  if( present( b ) ) then
    write (*,*) "Beide Parameter gegeben",  a, b
  else
    write (*,*) "Nur a gegeben", a
  end if

end subroutine ab
```

28.7. Module

Module ersetzen in Fortran 95 die FORTRAN 77-`COMMON`-Blöcke. In einem Modul können Variablen, Konstanten und auch Unterprogramme abgelegt werden. Diese können dann von verschiedenen Programmeinheiten angesprochen werden.

module modulname [implicit none] [save] Deklaration von Variablen, Konstanten [contains Unterprogramme] end module modulname

Das Einbinden eines Moduls in eineProgrammeinheit geschieht mittels der Anweisung

use modulname [, only liste]

`only` signalisiert, dass nur die in `liste` angegebenen Variablen oder Konstanten verwendet werden sollen.

Beispiel: **Fortran 90/95-Code (free source form)**

```
module m1
  implicit none
  save

  real, parameter :: PI=3.1415
  real :: a
end module m1

program bsp
  use m1
  implicit none

  a = 1.5
  write(*,*) 'Hauptprogramm 1: ', PI, a
  ! Ausgabe: Hauptprogramm 1:    3.141500       1.500000
  call sub
  write(*,*) 'Hauptprogramm 2: ', PI, a
  ! Ausgabe: Hauptprogramm 2:    3.141500       2.500000
end program bsp

subroutine sub
  use m1
  implicit none

  write(*,*)  'Unterprogramm: ', PI, a
  ! Ausgabe: Unterprogramm:    3.141500       1.500000
  a = 2.5
```

```
end subroutine sub
```

Das `save`-Statement in Modulen stellt sicher, dass die aktuellen Werte von Modulvariablen beim Wechsel zwischen den verschiedenen Programmeinheiten sicher erhalten bleiben.

28.8. Rekursiver Unterprogrammaufruf

In Fortran 95 können Unterprogramme (Funktionen, Subroutinen) auch rekursiv aufgerufen werden. Rekursion bedeutet, dass ein Unterprogramm sich selbst wieder aufruft (lat. recurrere oder en. recur ... wiederkehren, zurückkommen).

Beispiel: Berechnung von *n!* (vereinfacht) **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer :: zahl, ergebnis, fac

  write(*,*) "Gib eine Ganzzahl ein: "
  read(*,*) zahl

  ergebnis = fac(zahl)

  write(*,*) "Ergebnis ist: ", ergebnis
end program bsp


recursive function fac(n) result(zerg)
  implicit none
  integer, intent(in) :: n
  integer :: zerg

  ! Vereinfacht: Keine Überprüfung ob Überlauf, negative Zahl, etc.

  zerg = n

  if (n > 1) then
    zerg = n * fac(n-1)
```

```
  end if

  return
end function fac
```

Der Funktionskopf ist bei diesem Beispiel etwas anders gestaltet als üblich. Während der Intel-Fortran-Compiler auch ein

```
recursive integer function fac(n)
```

oder ein

```
integer recursive function fac(n)
```

problemlos akzeptiert, wirft der *gfortran*-Compiler bei diesen Varianten einen Syntaxfehler.

29. Ein- und Ausgabe

29.1. read

Die `read`-Anweisung dient dem Einlesen von Daten. Typisches Beispiel ist die Dateneingabe mittels Tastatur. Formal sieht eine `read`-Anweisung so aus:

read([unit=]*unit*, [fmt=]*fmt* [, iostat=*iostat*] [, advance=*advance*]) [eingabeliste]

- *unit* ... Nummer der Eingabeeinheit (ist systemabhängig), Sternoperator oder auch die einer Datei mittels `open`-Anweisung zugeordnete Nummer.
- *fmt* ... Anweisungsnummer zu einer `format`-Anweisung, Sternoperator oder Formatliste
- *iostat* ... read-Status
- *advance* ... siehe WRITE[1]

Listengesteuerte Eingabe auf der Standardeingabe (normalerweise die Tastatur):

```
read (*,*) a, b, c
```

1 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20FORTRAN%2095%3A%20EIN-{}%20UND%20AUSGABE%23WRITE%20

Alternativ kann das auch so geschrieben werden:

```
read (unit=*, fmt=*) a, b, c
```

Beim *Intel Fortran Compiler*, *gfortran* und *g95* ist auch `unit = 5` als `stdin` (Tastatur) vorbelegt. Das Einlesen aus Dateien und die Einstellung des Formates werden später erläutert.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
  integer :: i(5)
! Einlesen in ein Feld (unit ... Standardeingabe, fmt ...
 listengesteuert)
  read (*,*) i
! ...

end program bsp
```

Kurze Erläuterung zu `iostat`:

Wert	**Erläuterung**
0	kein Fehler
positiver Wert (systemabhängig)	Fehler
negativer Wert (systemabhängig)	*End Of File* und kein Fehler

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
  integer :: i, st
! Einlesen eines Wertes
  read (*, *, iostat=st) i

!  Ausgabe des IO-Status
   write (*,*) 'IO-Status:', st
```

Abb. 32

Die `write`-Anweisung dient der Datenausgabe. Typisches Beispiel ist die Anzeige von Daten auf dem Bildschirm. Formal sieht eine `write`-Anweisung so aus:

write([unit=]*unit*, [fmt=]*fmt* [, iostat=*iostat*] [, advance=*advance*]) [ausgabeliste]

- *unit* ... Nummer der Ausgabeeinheit (ist systemabhängig), Sternoperator oder auch die einer Datei mittels `open`-Anweisung zugeordnete Nummer.
- *fmt* ... Anweisungsnummer zu einer `format`-Anweisung, Sternoperator oder Formatliste
- *iostat* ... write-Status
- *advance* ... nur bei sequentieller formatierter I/O. Formatspezifizierer muss explizit gegeben sein.
  - `'no'` ... kein Vorschub des *file position pointers* zum nächsten Datensatz (z.B. kein Zeilenvorschub).

- `'yes'` ... mit Vorschub des *file position pointers* zum nächsten Datensatz (voreingestellt)

Listengesteuerte Ausgabe auf der Standardausgabe (normalerweise der Bildschirm):

```
write (*,*) a, b, c
```

Alternativ kann das auch so geschrieben werden:

```
write (unit=*, fmt=*) a, b, c
```

Beim *Intel Fortran Compiler*, *gfortran* und *g95* sind auch

- `unit`=0 als stderr (Bildschirm) und
- `unit`=6 als stdout (Bildschirm)

vorbelegt. Bezüglich `iostat` gilt auch hier der im vorigen Abschnitt kurz geschilderte Sachverhalt.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
  integer :: i(5) = (/ 5, 4, 1, -1, -7 /)
! ...
! Ausgabe der Feldwerte (unit ... Standardausgabe, fmt ...
 listengesteuert)
  write (*,*) i
! ...
end program bsp
```

Beispiel: `advance` **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
  character(10) :: ch
  integer :: st
  write(*, '(A)', advance='YES') "Hallo"
  write(*, '(A)', advance='YES') "Welt"
```

```
  write(*, *) "ABCDEFGHIJKLMN"
end program bsp
```

Ausgabe:

advance='YES'	**advance='NO'**
Hallo Welt ABCDEFGHI- JKLMN	HalloWelt ABCDEFGHI- JKLMN

Die Ausgabe in Dateien und die Einstellung des Formates werden etwas später erläutert.

29.3. Kürzer: print, read, write und Namenslisten

Für die listengesteuerte Ein- und Ausgabe existieren auch vereinfachte Formen. Für Eingaben wird wieder der read-Befehl verwendet, für Ausgaben gibt es die print-Anweisung.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
  integer               :: a, b
  real                  :: r
  complex               :: z
  character( len = 10 ) :: str
  read *, a, b, r, z, str
! Eingabe per Tastatur:
!   10, 30, 55.5, (10.8,7.0), Hallo

  print *, str, a, b, r, z
! Ausgabe am Bildschirm:
!   Hallo     10 30 55.5 (10.8,7.)
end program bsp
```

Bei mehrfachem Aufruf gleicher Ein- bzw. Ausgabeanweisungen kann durch Verwendung von Namenslisten der Programmcode kürzer gestaltet werden. Die Dateneingabe wird dadurch aber etwas komplizierter:

- eingeleitet wird die Eingabe durch ein &-Zeichen, unmittelbar gefolgt vom Namenslistenbezeichner
- danach folgen ein oder mehrere Leerzeichen
- es folgen die Zuweisungen von Werten zu den Variablennamen
- abgeschlossen wird die Eingabe durch einen Slash /

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
  integer               :: a, b
  real                  :: r
  complex               :: z
  character( len = 10 ) :: str
  namelist / LISTEA / a, b, r, z, str
  namelist / LISTEB / str, r, z
  read( *, nml = LISTEA )
! Eingabe per Tastatur:
!   &LISTEA b = 30, a = 10, r = 55.5, z = (10.8,7.0), str = "Hallo" /

  write( *, nml = LISTEB )
! Ausgabe auf dem Bildschirm (Intel 9.1):
!   &LISTEB
!   STR     = Hallo     ,
!   R       =   55.50000    ,
!   Z       = (10.80000,7.000000)
!   /
end program bsp
```

29.4. Formatierung

Abb. 33

Die Ein- und Ausgabeformatierung kann beeinflusst werden. Zu diesem Zweck gibt es die `format`-Anweisung.

... (..., fmt = *marke*, ...) ... *marke* format (*formatliste*)

Alternativ kann die Formatliste auch direkt in die `read`- oder `write`-Anweisung eingebunden werden

... (..., fmt = '(formatliste)', ...) ...

29.4.1. Formatlistenelemente

Formatspezifizierer	**Kommentar**
Ix[.z]	Ganzzahl mit einer Feldlänge von x Zeichen. z gibt die Mindestanzahl der auszugebenden Zeichen an (Feld wird, wenn nötig, mit führenden Nullen aufgefüllt).

Formatspezifizierer	Kommentar
Fx.y	Fixkommazahl mit einer Gesamtfeldlänge von x Zeichen. y ist die Anzahl der Nachkommastellen (Vorzeichen und Dezimalpunkt müssen in der Gesamtfeldlänge berücksichtigt werden).
Ex.y	Gleitkommazahl mit einer Gesamtfeldlänge von x Zeichen. y ist die Anzahl der Nachkommastellen. (Vorzeichen, Dezimalpunkt und die Zeichen für den Exponenten müssen in der Gesamtfeldlänge berücksichtigt werden).
Dx.y	-"-
A	Eine Zeichenkette.
Ax	Eine Zeichenkette mit x Zeichen.
Lx	Ein logischer Wert, T bzw. F
xX	x Leerzeichen.
/	Zeilenvorschub

Obige Tabelle der Formatlistenelemente ist nicht vollständig. Fortran kennt noch weitere Formatierungsmöglichkeiten. Die Ausgabe erfolgt normalerweise rechtsbündig. Reicht die Gesamtfeldlänge bei numerischen Werten nicht aus, so werden anstelle einer Zahl Sternchen angezeigt.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer :: a

  a = 999
  write(*, 3333) a
  ! Ausgabe: 999

  a = -999
  write (*, 3333) a
  ! Ausgabe: ***

3333 FORMAT (I3, /, /)
  ! / ... nach jeder 3333-write-Anweisung werden zwei Leerzeilen
 eingefügt
end program bsp
```

Alternativ könnte die Formatliste auch so in die `write`-Anweisung eingebaut werden:

```
write(*, '(I3, /, /)') a
```

Weitere Formatierungsbeispiele:

Code	**Ausgabe**
WRITE(*, 999) 1234 WRITE(*, 999) 1234567 WRITE(*, 999) 1234567890 999 FORMAT(I9.6)	001234 1234567 *********
WRITE(*, 999) 555.6666 WRITE(*, 999) +5.6 WRITE(*, 999) -55.666E7 WRITE(*, 999) -55555.666 999 FORMAT(F9.3)	555.667 5.600 ********* *********

Code	Ausgabe
WRITE(*, 999) 555.6666 WRITE(*, 999) +5.6 WRITE(*, 999) -55.666E7 WRITE(*, 999) -55555.666 999 FORMAT(E9.3)	0.556E+03 0.560E+01 -.557E+09 -.556E+05
WRITE(*, 999) 'Hallo' WRITE(*, 999) 'ABCDEFGHIJKL' WRITE(*, 888) 'ABCDEFGHIJKL' 888 FORMAT(A) 999 FORMAT(A10)	Hallo ABCDEFGHIJ ABCDEFGHIJKL
WRITE(*, *) 'FORTRAN', '77' WRITE(*, 999) 'FORTRAN', '77' 999 FORMAT(A, 1X, A)	FORTRAN77 FORTRAN 77
WRITE(*, 888) 'FORTRAN', '77' WRITE(*, 999) 'FORTRAN', '77' 888 FORMAT(A, T3, A) 999 FORMAT(A, T20, A)	FO77RAN FORTRAN 77
WRITE(*, 999) 'FORTRAN', '77' 999 FORMAT(A, /, A)	FORTRAN 77
WRITE(*, 999) 34.56 WRITE(*, *) 34.56 C SP ... Sign Plus (+) 999 FORMAT(SP, F12.3)	+34.560 34.56

29.4.2. Wiederholung von Formatteilen

Beispiel:

```
  write(*, 100) 'abc', 10.3, 'xxx', 23.4
100 format (2(A3, F6.1))
```

29.4.3. write etwas anders

Beispiel:

```
  write (*, 100)
100 format ('Hallo', X, 'Welt!')
```

29.4.4. Dynamische Mehrfach Formatierung

Formatierungsanweisungen können auch als String bearbeitet werden, indem die z.B. die Anzahl der auszugebenden Variablen per **write**-Befehl in die Formatierung schreibt. Mehrfachformatierung - Beispiel:

```
character(4) :: formatierung
integer, dimension(1:4) :: einsen = 1
integer :: anzahl
anzahl = 4
formatierung = '( I)'
write(formatierung(2:2), '(I1)') anzahl
write(*, formatierung) einsen
```

29.5. Dateien

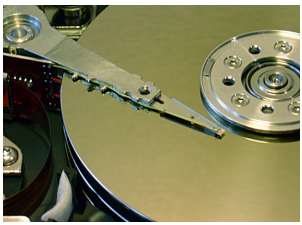


Abb. 34

29.5.1. Datensatz

Datensätze können in folgender Form auftreten:

- Formatierter Datensatz: Textdatensatz
- Unformatierter Datensatz: Datensatz in einer maschineninternen Form.
- Dateiendesatz

29.5.2. Datei

Für Fortran ist alles eine Datei, das durch `read` oder `write` bearbeitbar ist.

Zugriffsmethoden:

- Sequentieller Zugriff: Lesen ab Beginn der Datei (file) und dann immer den nächsten Datensatz einlesen. Geschrieben wird jeweils ans Dateiende. Auf interne Dateien kann nur sequentiell zugegriffen werden.

- Direkter Zugriff: Bearbeiten in beliebiger Reihenfolge durch Angabe der Satznummer.
- Binärer Zugriff: Bearbeiten von Dateien, die binäre Daten enthalten, z. B. Bilder von CCD-Kamera, Scilab/Matlab `save`-Dateien

Dateitypen:

- Externe Datei: Eine konventionelle Datei
- Interne Datei: character-Variable oder -Feld.

Dateien haben im Betriebsystem einen Dateinamen. In Fortran wird eine Datei über eine Dateinummer (unit) angesprochen. Die Zuordnung erfolgt mit dem Befehl `open`.

29.5.3. open

Zum Öffnen einer externen Datei dient die `open` -Anweisung.

```
open (liste)
```

mit folgender *liste*

Element	**Kommentar**
[unit =] x	x ist eine Dateinummer (Ganzzahl, sollte über 10 liegen, da oft Nummern unter 10 fix zugeordnet sind, z.B. der Standardein-, ausgabe).
file = x	x ist der externe Dateiname
iostat = x	x ist 0 wenn open fehlerfrei ausgeführt wurde, ansonsten eine systemabhängige Fehlernummer

Element	Kommentar
status = x	Dateistatus: 'old' ... Datei existiert bereits 'new' ... Datei wird neu erzeugt 'scratch' ... namenlose temporäre Datei 'unknown' ... System bestimmt Dateistatus selbst 'replace' ... der Inhalt einer bereits vorhandenen Datei wird gelöscht.
access = x	Zugriffsmethode: 'sequential' ... Sequentielle Datei 'direct' ... direkter Zugriff 'stream' ... binärer Zugriff
position = x	Den Dateisatzzeiger beim Öffnen der Datei an eine bestimmte Position setzen. ('asis', 'rewind', 'append')
form = x	Format: 'formatted' oder 'unformatted'
action = x	'read' ... nur Lesezugriff 'write' ... nur Schreibzugriff 'readwrite' ... Lesen und Schreiben
recl = x	Datensatzlänge (positive Zahl, access='direct', in Bytes)
err = x	Im Fehlerfall Sprung zur Marke x

Element	Kommentar
blank = x	'null' oder 'zero' (nur für form='formatted')
delim = x	'apostrophe' 'quote' 'none'
pad = x	'yes' oder 'no' (nur für form='formatted')

Eingestellte Vorgabewerte sind:

- status = 'unknown'
- position = 'asis'
- access = 'sequential'
- form = 'formatted'

Wird `access='direct'` gesetzt, so gilt `form='unformatted'` als Vorgabewert.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer :: stat
  character(80) :: str

  open(20, file='/tmp/testdatei.txt', iostat=stat)

  if(stat == 0) then
    write(*,*) 'Das Öffnen der Datei war ein voller Erfolg'

    do
      read(20, '(A)', iostat=stat) str
      ! Bei EOF wird stat /= 0
      if (stat /= 0) exit
      write(*,*) str
    end do
  else
    write(*,*) 'Datei konnte nicht geöffnet werden'
  end if
```

```
  close(20)
end program bsp
```

29.5.4. close

Geschlossen wird die Verbindung zur externen Datei mit dem `close`-Befehl.

close (liste)

liste:

Element	**Kommentar**
[unit =] x	wie bei open
iostat = x	wie bei open
err = x	wie bei open
status = x	'keep' ... Datei erhalten (voreingestellt) 'delete' ... Datei löschen

29.5.5. Lesen und Schreiben

Gelesen oder geschrieben wird mit den bereits bekannten `read`- und `write`-Anweisungen.

Element	**Kommentar**
[unit =] x	Dateinummer bzw. CHARACTER-Variable oder Feld (interne Datei)

Element	Kommentar
[fmt =] x	siehe FORMATIERUNG[2]
[nml] = x	x ... namelist-group-name
rec = x	Datensatznummer bei Direktzugriff (siehe Abschnitt DIREKTZUGRIFF[3])
iostat = x	wie bei `read`[4]
err = x	Bei Fehler Sprung zur Anweisungsnummer x
end = x	Bei Dateiende Sprung zur Anweisungsnummer x (nicht erlaubt bei Direktzugriff, nicht bei write)
advance = x	'yes' oder 'no'
eor = x	Bei *End of Record* Sprung zur Marke x (nicht bei write)
size = x	x ... Zeichenzähler (nicht bei write, advance='no')

Es existiert eine Menge von Einschränkungen, wann welche Elemente erlaubt sind, bzw. welche nur kombiniert auftreten sollen, z.B.

- wenn der `rec`-Spezifizierer Verwendung findet, dann darf kein `end`-Element angegeben werden
- Bei Dateneingabe nur dann ein `size`-Spezifizierer, wenn `advance='no'` gesetzt ist.

2 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20FORTRAN%2095%3A%20EIN-{}%20UND%20AUSGABE%23FORMATIERUNG

3 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20FORTRAN%2095%3A%20EIN-{}%20UND%20AUSGABE%23DIREKTZUGRIFF

4 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20FORTRAN%2095%3A%20EIN-{}%20UND%20AUSGABE%23READ

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  character (len = 80) :: a
  integer              :: st = 0

  open (20, file='/tmp/testdatei.txt', status='OLD', iostat=st)

  if (st /= 0) then
    stop "open-Fehler!"
  end if

  ! Aus Datei lesen
  do
    read (20, 888, iostat=st) a
    ! Auf Standardausgabe schreiben

    if (st == 0) then
      write (*, 888) a
    else if (st > 0) then
      write (*,*) "read-Fehler!"
      exit
    else if (st < 0) then
      exit
    end if
  end do

  close(20)

 888  format(A)
end program bsp
```

Direktzugriff

OPEN:

Element	Kommentar
access = x	x ... 'DIRECT'

Element	Kommentar
recl = x	x ... Datensatzlänge (positive Zahl, access='DIRECT', in Bytes bzw. bei formatierten Dateien in Characters)
fmt = x	x ... Formatangabe (wird eine Datei 'FORMATTED' geöffnet, dann muss auch eine konkrete Formatliste angegeben werden, ansonsten tut's auch der Sternoperator)

READ/WRITE:

Element	Kommentar
REC = x	x ... Satznummer bei Direktzugriff

Beispiel: Gegeben ist die Textdatei `/tmp/testdatei.txt` mit dem Inhalt

```
Die WRITE-Anweisung dient der Datenausgabe aus einem FORTRAN-Programm
auf ein
externes Gerät. Typisches Beispiel ist die Anzeige von Daten auf dem
Bildschirm.
Formal sieht eine WRITE-Anweisung so aus:
```

Fortran 90/95-Code (free source form)

```
program bsp
  implicit none

  character (len = 10) :: c
  integer              :: st
```

```
  open (20, file='/tmp/testdatei.txt', status='OLD', form='FORMATTED',
 &
            access='DIRECT', recl=15, iostat=st)
  if (st /= 0) then
    stop "open-Fehler!"
  end if
  read (20, fmt='(A)', rec=4, iostat=st) c
  if (st /= 0) then
    write (*,*) "read-Error"
  else
    write (*,*) c
  end if
  close (20)
end program bsp
```

Ausgabe:

```
 s einem FO
```

29.5.6. Positionieren bei sequentiellen Dateien

Datensatzzeiger um einen Datensatz zurücksetzen:

```
backspace ([unit=]x [,iostat=y] [,err=z])
```

Positionieren an den Dateibeginn:

```
rewind ([unit=]x [,iostat=y] [,err=z])
```

Schreiben eines Dateiendsatzes:

```
endfile ([unit=]x [,iostat=y] [,err=z])
```

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
```

```
  implicit none

  character (len = 100), dimension(3) :: c
  integer                             :: st = 0

  open (20, file='/tmp/testx.txt', status='NEW', iostat=st)
  call checkStatus(st, "open-")
  write (20,*) 'Das ist eine Testdatei'
  write (20,*) 'Dies ist Zeile 2 der Testdatei'
  write (20,*) 'Jenes die Zeile 3 der Testdatei'
  write (20,*) "Jetzt ist's aber genug"
  endfile (20)
  rewind (20, iostat=st)
  call checkStatus(st, "rewind-")

  read (20, fmt=555, iostat=st) c
  call checkStatus(st, "read-")

  write (*, fmt=555) c
  backspace (20, iostat=st)
  call checkStatus(st, "backspace-")

  read (20, fmt=555, iostat=st) c(1)
  call checkStatus(st, "read-")

  write (*, fmt=555) c(1)

  close (20)

 555 format (A)
end program bsp
subroutine checkStatus(st, ch)
  integer,       intent (in) :: st
  character (*), intent (in) :: ch

  if (st /= 0) then
    close(20)
    write (*,*) ch // "Fehler!"
    stop
  end if
end subroutine checkStatus
```

Ausgabe:

```
Das ist eine Testdatei

Dies ist Zeile 2 der Testdatei

Jenes die Zeile 3 der Testdatei

Jenes die Zeile 3 der Testdatei
```

29.5.7. inquire

Die Anweisung `inquire` dient der Abfrage einiger Eigenschaften von Dateien oder I/O-Units.

inquire (file = *x, liste*)

mit x ... Dateiname (inkl. Pfad)

inquire ([unit =] *x, liste*)

mit x ... Nummer der I/O-Unit.

liste:

Element	**Kommentar**
access = x	x: • 'SEQ' ... sequentieller Dateizugriff • 'DIRECT' ... Direktzugriff

Element	Kommentar
action = x	x: • 'READ' • 'WRITE' • 'READWRITE' • 'UNDEFINED'
blank = x	x: • 'NULL' • 'ZERO'
delim = x	x: • 'APOSTROPHE' • 'QUOTE' • 'NONE' • 'UNDEFINED'
direct = x	x: • 'YES' ... Direktzugriff • 'NO' ... kein Direktzugriff für diese Datei erlaubt • 'UNKNOWN' ... unbekannt
err = x	Bei Fehler Sprung zur Anweisungsnummer x

Element	Kommentar
exist = x	x: • .TRUE. ... Datei existiert • .FALSE. ... Datei existiert nicht
form = x	x: • 'FORMATTED' ... Datei geöffnet für formatierte Datensätze • 'UNFORMATED' ... Datei geöffnet für unformatierte Datensätze • 'UNDEFINED'
formatted = x	x: • 'YES' ... formatierte Datensätze sind erlaubt • 'NO' ... formatierte Datensätze sind nicht erlaubt • 'UNKNOWN' ... unbekannt
iostat = x	x ist 0 wenn OPEN fehlerfrei ausgeführt wurde, ansonsten eine systemabhängige positive Fehlernummer

Element	Kommentar
name = x	Der Dateiname wird der Zeichenketten-Variablen x zugewiesen. Hat die Datei keinen Namen, dann ist ist das Ergebnis undefiniert.
named = x	x: • .TRUE. ... Datei besitzt Namen • .FALSE. ... Datei besitzt keinen Namen
nextrec = x	x ... Nummer des nächsten Datensatzes
number = x	x ... Nummer der mit einer externen Datei verbundenen I/O-Unit.
opened = x	x: • .TRUE. ... Datei ist geöffnet • .FALSE. ... Datei ist nicht geöffnet
pad = x	x: • 'YES' • 'NO'

Element	Kommentar
position = x	x: • 'REWIND' • 'ASIS' • 'APPEND' • 'UNDEFINED'
read = x	x: • 'YES' • 'NO' • 'UNKNOWN'
readwrite = x	x: • 'YES' • 'NO' • 'UNKNOWN'
recl = x	x ... Datensatzlänge bei Direktzugriff
sequential = x	x: • 'YES' ... sequentiell • 'NO' ... nicht sequentiell • 'UNKNOWN' ... unbekannt

Element	Kommentar
unformatted = x	x: • 'YES' ... unformatierte Datensätze sind erlaubt • 'NO' ... unformatierte Datensätze sind nicht erlaubt • 'UNKNOWN' ... unbekannt
write = x	x: • 'YES' • 'NO' • 'UNKNOWN'

Beispiel: Datei vorhanden? **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
  logical :: l
  integer :: st

  inquire (file='/tmp/testdatei.txt', exist=l, iostat=st)
  if (st == 0) then
    write (*,*) "Datei existiert?", l
  else
    write(*,*) "Fehler!"
  end if
!  wenn Datei existiert:       Datei existiert?  T
!  wenn Datei nicht existiert: Datei existiert?  F
!  wenn aus irgendeinem ein inquire-Fehler auftrat: Fehler!
end program bsp
```

Beispiel: Infos zu einer geöffneten Datei **Fortran 90/95-Code (free source form)**

```
program bsp
   implicit none

   logical        :: ex
   character (15) :: di, fo, ac, se
   integer        :: nu, st

   open (25, file='/tmp/testdatei.txt', status='old', iostat=st)

   if(st /= 0) stop "open-Fehler!"

   inquire (25, exist = ex, direct = di, sequential = se, formatted =
 fo, &
                access = ac, number = nu, iostat=st)
   if(st == 0) then
     write (*,*) 'EXIST? ', ex
     write (*,*) 'DIRECT? ', di
     write (*,*) 'SEQUENTIAL? ', se
     write (*,*) 'FORMATTED? ', fo
     write (*,*) 'ACCESS? ', ac
     write (*,*) 'NUMBER? ', nu
   else
     write (*,*) "inquire-Fehler!"
   end if

   close(25)

! Ausgabe, z.B.
!    EXIST?  T
!    DIRECT? YES
!    SEQUENTIAL? YES
!    FORMATTED? YES
!    ACCESS? SEQUENTIAL
!    NUMBER?           25
end program bsp
```

29.5.8. Interne Dateien

- Interne Dateien sind vom Datentyp `character` (Zeichen oder Zeichenketten)

- Das Lesen aus bzw. das Schreiben in interne Dateien erfolgt immer sequentiell

Beispiel: Schreiben in eine interne Datei **Fortran 90/95-Code (free source form)**

```
program bsp
  character(15) :: ch
  real          :: r = 12.5678

! Interne Datei "ch"
  write (ch, *) r

  write (*,*) 'r lexikalisch groesser als Buchstabe "A"? ', lge(ch,
 'A')
end program bsp
```

Beispiel: Lesen aus einer internen Datei **Fortran 90/95-Code (free source form)**

```
program bsp
  character(15) :: ch = '12.5678'
  real          :: r

! Interne Datei "ch"
  read (ch, '(F15.5)') r

  write (*,*) 'r    = ', r
  write (*,*) 'r**2 = ', r**2
end program bsp
```

30. Zeiger

30.1. Was sind Zeiger?

ZEIGER (INFORMATIK)[1]

30.2. Zeiger in Fortran 95

In Fortran 95 werden Zeiger durch Zufügen des Attributes `pointer` bei der Deklaration von Variablen erzeugt.

datentyp, pointer :: *variable*

Ein so deklarierter Zeiger kann auf andere Zeiger oder auf mittels `target` gekennzeichnete Variablen verweisen.

datentyp, target :: *variable*

Die Zeigerzuordnung erfolgt durch das Symbol

=>

1 HTTP://DE.WIKIPEDIA.ORG/WIKI/ZEIGER%20%28INFORMATIK%29

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  real, pointer :: ptr
  real, target :: trg

  trg = 5.5
  ptr => trg
  write(*,*) ptr
  ! Ausgabe: 5.50000
  ! Zeiger werden bei Bedarf automatisch dereferenziert
end program bsp
```

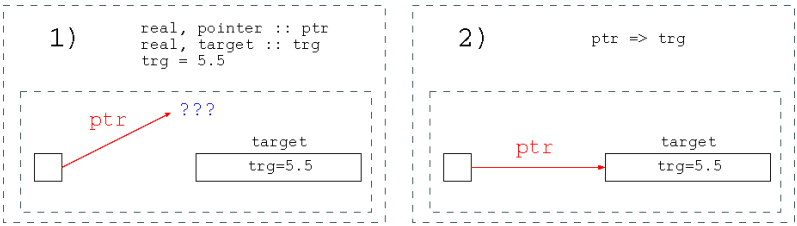


Abb. 35

30.3. Assoziationsstatus

Ein Zeiger kann einen der folgenden Assoziationzustände annehmen:

- undefiniert (dangling)
- nicht zugeordnet (disassociated, null)
- zugeordnet (associated)

Der Zuordnungsstatus eines Zeigers ist unmittelbar nach der Deklaration *undefiniert*. Mittels *zeiger* `=> null()` oder

`nullify(`*zeiger*`)` kann ein Zeiger auf einen *nicht zugeordneten* Status gesetzt werden. Verweist ein Zeiger auf einen anderen zugeordneten Zeiger oder ein Target, so ist sein Zustand *zugeordnet*.

Der Assoziationsstatus eines Zeigers lässt sich über die Funktion

```
associated (zeiger [, ziel])
```

abfragen. Sinnvoll ist eine derartige Abfrage nur dann, wenn der Zuordnungsstatus nicht undefiniert ist.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer, pointer       :: ptr1 => null()
  character(20), pointer :: ptr2
  character(20), target  :: str

  str = "Hallo, Welt!"
  ptr2 => str

  write(*,*) associated(ptr1)
  ! Ausgabe: F

  write(*,*) associated(ptr2)
  ! Ausgabe: T

  write(*,*) associated(ptr2, ptr1)
  ! Ausgabe: F

  write(*,*) associated(ptr2, str)
  ! Ausgabe: T
end program bsp
```

30.4. Speicherplatz dynamisch reservieren und freigeben

Für normale Variablen läuft die Speicherplatzverwaltung automatisch ab. Bisher wurden Zeiger immer solchen normalen (Target)Variablen, für die bereits Speicherplatz reserviert war, zugeordnet. Aber auch für Zeiger selbst kann Speicherplatz reserviert werden. Bei der Zeigerdeklaration ist der Zeigerstatus undefiniert oder nicht zugeordnet. Eine Wertzuweisung an eine solche Zeigervariable würde zur Laufzeit einen Speicherzugriffsfehler ergeben. Die Funktion

```
allocate (zeiger1, [zeiger2, ...] [,stat=integervar])
```

reserviert in Abhängigkeit des Zeiger-Datentyps Speicherplatz für die einzelnen Zeiger. Die Funktion

```
deallocate (zeiger1, [zeiger2, ...] [,stat=integervar])
```

gibt diesen Speicherplatz wieder frei.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer, pointer :: ptr1 => null(), ptr2 => null()
  integer :: status

  allocate(ptr1, stat = status)
  ptr1 = 2222

  write (*,*) "Status = ", status
  ! Wenn status = 0, dann wurde erfolgreich Speicherplatz reserviert

  write (*,*) ptr1
  ! Ausgabe: 2222
```

```
  ptr2 => ptr1
  ptr1 = 5555

  write (*,*) ptr1, ptr2
  ! Ausgabe: 5555   5555

  deallocate(ptr1)
end program bsp
```

30.5. Zeiger und Felder

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer, dimension(:), pointer   :: ptr => null()
  integer, dimension(5:10), target :: arr = (/55, 66, 77, 88, 99,
 111/)

  ptr => arr
  write(*,*) ptr
  ! Ausgabe:  55    66    77    88    99    111

  ptr => arr(7:)
  write(*,*) ptr
  ! Ausgabe:  77    88    99    111
end program bsp
```

Beispiel: "ragged array"

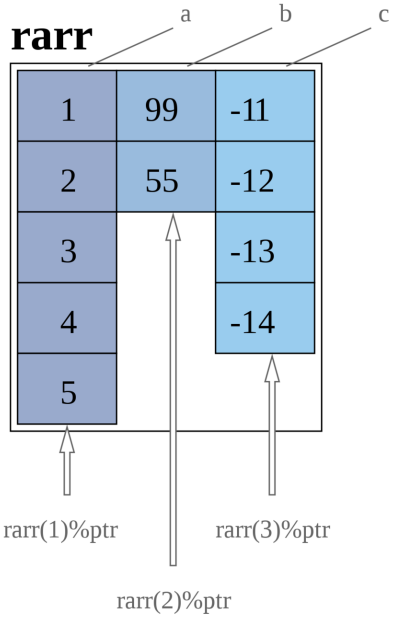


Abb. 36

Fortran 90/95-Code (free source form)

```
program bsp
  implicit none

  type element
    integer, dimension(:), pointer :: ptr
  end type element
  integer, dimension(5), target :: a = (/ 1, 2, 3, 4, 5 /)
  integer, dimension(2), target :: b = (/ 99, 55 /)
  integer, dimension(4), target :: c = (/ -11, -12, -13, -14 /)
```

```
  integer                       :: i
  type( element ), dimension(3) :: rarr

  rarr(1)%ptr => a
  rarr(2)%ptr => b
  rarr(3)%ptr => c
  do i = 1,3
    write (*,*) "rarr(", i, "): ", rarr(i)%ptr
  end do

! Ausgabe
!   rarr( 1 ):  1 2 3 4 5
!   rarr( 2 ):  99 55
!   rarr( 3 ):  -11 -12 -13 -14
end program bsp
```

30.6. Verkettete Listen

Beispiel: Einfach verkettete Liste (LIFO) **Fortran 90/95-Code (free source form)**

```
module m1
  implicit none

  type node
    integer :: id
    character(5) :: value
    type(node), pointer :: next => null()
  end type

  type(node), pointer :: first => null()

  private ! Auf alle nachfolgenden Anweisungen kann von aussen nicht
 zugegriffen werden
  public :: add_node, write_all, free_all ! Auf die Subroutinen
 add_node, write_all und free_all
                                ! kann von aussen explizit zugegriffen
 werden,
                                ! jedoch nicht auf innerhalb der
 Subroutinen
                                ! deklarierte Datentypen
  contains
    subroutine add_node(id, str)
```

```
  implicit none

  integer, intent(in) :: id
  character(5), intent(in) :: str

  type(node), pointer :: new, tmp

  ! Speicher reservieren
  allocate(new)

  ! Werte setzen
  new%id = id
  new%value = str

  ! Am Beginn der Liste einfügen
  if (associated(first) .eqv. .FALSE.) then
    first => new
  else
    tmp => first
    first => new
    first%next => tmp
  end if

end subroutine add_node


subroutine write_all()
  implicit none

  type(node), pointer :: tmp

  tmp => first

  do
    if (associated(tmp) .eqv. .FALSE.) exit

    write(*,*)tmp%id, tmp%value
    tmp => tmp%next
  end do
end subroutine write_all
subroutine free_all()
  implicit none

  type(node), pointer :: tmp

  do
      tmp => first
```

```
        if (associated(tmp) .eqv. .FALSE.) exit

        first => first%next
        deallocate(tmp)
      end do
    end subroutine free_all
end module m1


program bsp
  use m1
  implicit none

  call add_node (1, "AAAAA")
  call add_node (2, "BBBBB")
  ! ...
  call add_node (150, "ZZZZZ")

  call write_all
  ! Ausgabe:
  !   150 ZZZZZ
  !   2 BBBBB
  !   1 AAAAA
  call free_all ! Die verkettete Liste wird wieder freigegeben
end program bsp
```

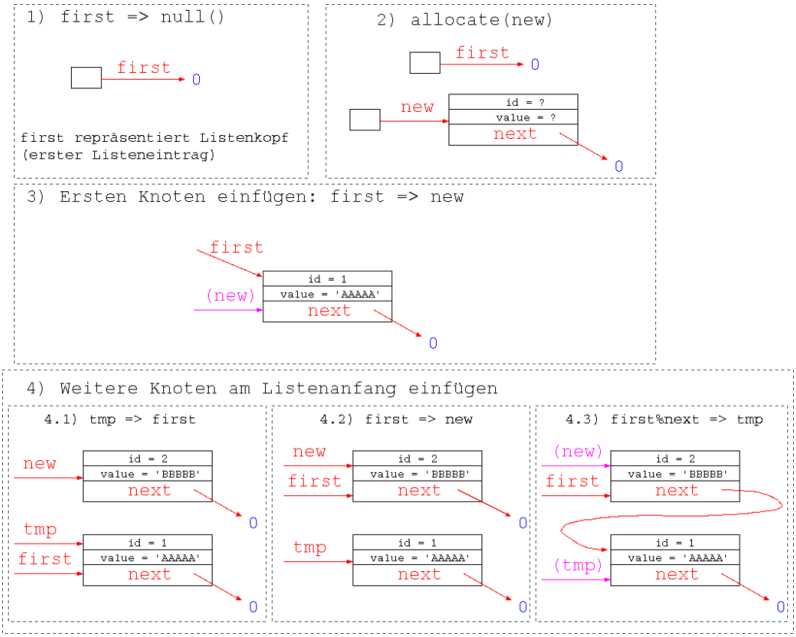

1) first => null()
first
0
first repräsentiert Listenkopf
(erster Listeneintrag)
2) allocate(new)
first
0
new
id = ?
value = ?
next
0
3) Ersten Knoten einfügen: first => new
first
(new)
id = 1
value = 'AAAAA'
next
0
4) Weitere Knoten am Listenanfang einfügen
4.1) tmp => first
new
id = 2
value = 'BBBBB'
next
0
tmp
first
id = 1
value = 'AAAAA'
next
0
4.2) first => new
new
first
id = 2
value = 'BBBBB'
next
0
tmp
id = 1
value = 'AAAAA'
next
0
4.3) first%next => tmp
(new)
first
id = 2
value = 'BBBBB'
next
(tmp)
id = 1
value = 'AAAAA'
next
0

Abb. 37

31. Vektoren- und Matrizenrechnung

In Fortran95 können einige elementare Vektoren- und Matrizenoperationen sehr einfach ausgeführt werden.

Beispiel: Addition und Subtraktion von Vektoren (Matrizen) **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  real, dimension(3) :: a = (/3, 2, -5/), b = (/1, -3, -1/)

  write(*,*) "a+b: ", a+b
  ! Ausgabe:  a+b:  4. -1. -6.

  write(*,*) "a-b: ", a-b
  ! Ausgabe: a-b:  2. 5. -4.
end program bsp
```

Beispiel: Multiplikation eines Vektors (einer Matrix) mit einem Skalar **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  real, dimension(3) :: a = (/3, 2, -5/)

  write(*,*) "3.5*a: ", 3.5*a
  ! Ausgabe: 3.5*a:  10.5 7. -17.5
end program bsp
```

Beispiel: Skalarprodukt **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  real, dimension(3) :: a = (/3, 2, -5/), b = (/1, -3, -1/)
  real               :: dot_product

  write(*,*) "a.b: ", dot_product(a, b)
  ! Ausgabe: 2
end program bsp
```

Beispiel: Euklidische Norm eines Vektors $|\mathbf{x}| = \sqrt{\mathbf{x}^2} = \sqrt{\sum_{i=1}^{n} x_i^2}$ **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  real, dimension(3) :: x = (/3, 2, -5/)
  real               :: x_n

  ! Norm des Vektors x
  x_n = sqrt(sum(x**2))

  write(*,*) "Die Norm des Vektors (", x, ") beträgt: ", x_n
  ! Ausgabe: Die Norm des Vektors ( 3.0  2.0 -5.0 ) beträgt:  6.164414
end program bsp
```

Beispiel: Matrizenmultiplikation **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  real, dimension(3,2) :: A = reshape( (/3., 2., 1., 1., 1., 2.5/),
 (/3, 2/) )
  real, dimension(2,2) :: B = reshape( (/1., -3., -1., 5./), (/2, 2/)
 )
  real, dimension(3,2) :: C

  C = matmul(A, B)

  write(*,*) "Matrix C ="
  write(*,*) C(1, :)
  write(*,*) C(2, :)
  write(*,*) C(3, :)
  ! Ausgabe: Matrix C =
```

```
  !           0.000000       2.000000
  !           -1.000000      3.000000
  !           -6.500000      11.50000
end program bsp
```

Beispiel: Transponierte Matrix **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  real, dimension(2,2) :: A = reshape( (/3., 2., 1., -1.5/), (/2, 2/)
 ), AT

  AT = transpose(A)

  write(*,*) "A ="
  write(*,*) A(1, :)
  write(*,*) A(2, :)
  ! Ausgabe: A =
  ! 3.000000       1.000000
  ! 2.000000      -1.500000

  write(*,*) "AT ="
  write(*,*) AT(1, :)
  write(*,*) AT(2, :)
  ! Ausgabe: AT =
  ! 3.000000       2.000000
  ! 1.000000      -1.500000
end program bsp
```

32. Systemroutinen

32.1. Datum und Zeit

Die Subroutine

```
date_and_time(datum,zeit)
```

liefert die Systemzeit in der Form YYYYMMTT und HHMMSS.SSS zurück. `datum` und `zeit` sind Character von mindestens 8 bzw. 10 Zeichen Länge.

Die vom Programm verbrauchte Rechenzeit (Prozessorzeit) in Sekunden liefert die Subroutine (`zeit` ist vom Typ Real)

```
cpu_time(zeit)
```

32.2. Zufallszahlen

Die Subroutine

```
random_number(r)
```

schreibt gleichverteilte Zufallszahlen im Intervall [0,1) in eine Variable `r` vom Typ Real (Skalar oder Feld). Mit der Subroutine

```
random_seed()
```

kann der Zufallszahlengenerator (zufällig) initialisiert werden.

32.3. Kommandozeilenargumente

Nicht unbedingt Standard, aber bei etlichen Fortran-Compilern doch als Standardfunktion implementiert, sind die Prozeduren `iargc` und `getarg` zum Erfragen der beim Programmstart mitgegebenen Kommandozeilenargumente.

Die Funktion

```
i = iargc()
```

liefert die Anzahl der Kommandozeilenargumente. Der Programmname selbst wird dabei nicht mitgezählt.

Die Subroutine

```
getarg(i, c)
```

liefert den Wert eines bestimmten Kommandozeilenargumentes. `i` gibt die Position vor (0 ...Programmname, 1 ... 1.Argument, etc.). Der Parameter `c` ist vom Typ Character. Dort findet sich nach Abarbeitung der Subroutine der zu `i` gehörende Wert des Kommandozeilenargumentes.

33. Von der modularen zur objektorientierten Programmierung

33.1. Module im Detail

Die Bezeichnung des Schlüsselworts `module` weist schon darauf hin, dass Fortran 90/95 eine modulare Softwareentwicklung ermöglicht.

33.1.1. Modulare Programmierung

Was ist modulare Programmierung? MODULARE PROGRAMMIERUNG[1]

Das Modul-Konzept in Fortran 90/95 unterstützt diesen Ansatz vollständig. Das `module`-Konstrukt gliedert sich schematisch so

```
module ...

    Datenbereich
```

1 HTTP://DE.WIKIPEDIA.ORG/WIKI/MODULARE%20PROGRAMMIERUNG

Methodenbereich

end module ...

Vor dem Datenbereich können noch einige Deklarationen (`implicit none, save`, etc.) eingefügt sein. Der Methodenbereich wird durch das Schlüsselwort `contains` angekündigt oder als Interface deklariert. Aber prinzipiell gilt, dass zusammengehörende Daten und die dazugehörenden Methoden (Unterprogramme) in einem Modul zusammengefasst werden.

33.1.2. Zugriffsteuerung

Durch Angabe des Schlüsselwort `private` läßt sich die Sichtbarkeit von Datenelementen einschränken. Auf solcherart deklarierte Variablen läßt sich außerhalb des Moduls nicht zugreifen. `public` erlaubt den Zugriff auf entsprechend deklarierte Variablen auch von außerhalb. Zweiteres ist Standardverhalten und das Schlüsselwort `public` muss somit nicht explizit angegeben werden.

Beispiel: **Fortran 90/95-Code (free source form)**

```
module mod_bsp
  implicit none
  save
! Datenbereich
  real, private :: x = 1.2
  real          :: y = 9.8

  contains
! Methodenbereich
    real function addX (a)
      real, intent (in) :: a
      addX = x + a
    end function addX
```

```
end module mod_bsp
program bsp
  use mod_bsp
  implicit none

  write (*,*) "Ergebnis1 = ", addX (2.1)
  write (*,*) "Ergebnis2 = ", y + 2.1
! Ausgabe:
!    Ergebnis1 =    3.300000
!    Ergebnis2 =    11.90000

! Folgendes geht nicht -> Fehlermeldung:
! write (*,*) "Ergebnis3 = ", x + 2.1
!   1)  x ist private und somit außerhalb des Moduls nicht bekannt
!   2) auch im Hauptprogramm selbst ist kein x deklariert (implicit
 none,
       es wäre ohnehin nicht die gleiche Variable wie im Modul
 mod_bsp)
end program bsp
```

Rein formal läßt sich der Zugriff auf die Daten und/oder Methoden eines Moduls (module procedures) auch auf andere Arten einschränken, z.B.

```
module mod_bsp
  implicit none
  save

  private :: x  ! , ....

  real :: x = 1.2, y = 9.8
! ...
```

oder

```
module mod_bsp
  implicit none
  save

  private
```

```
  public :: y, addX  ! , ...

  real :: x = 1.2, y = 9.8
! ...
```

33.1.3. Datenkapselung, COMMONS-Ersatz: Module als Datenbereich

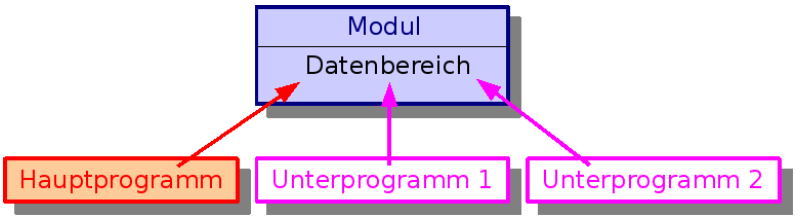


Abb. 38

Module als reinen Datenbereich zu nutzen ist vor allem dann interessant, wenn aus mehreren Programmeinheiten auf die gleichen Daten zugegriffen werden muss, jedoch die zugreifenden Unterprogramme nicht wirklich modulspezifisch sind und dementsprechend nicht als Bestandteil des Moduls angelegt werden.

Beispiel: **Fortran 90/95-Code (free source form)**

```
module konstanten
  real, parameter :: PI = 3.141593
  real, parameter :: E =  2.718282
end module konstanten


! Hauptprogramm
program bsp
  use konstanten
  implicit none
```

```
  real :: kreisflaeche

  write(*,*) 'PI = ', PI
  write(*,*) 'E = ',  E
  write(*,*) 'Kreisflaeche fuer r=2.1 = ', kreisflaeche(2.1)
  call calcPiMalE
end program bsp


! Unterprogramm 1
real function kreisflaeche(r)
  use konstanten
  implicit none
  real, intent (in) :: r
  kreisflaeche = r**2 * PI
end function kreisflaeche
! Unterprogramm 2
subroutine calcPiMalE()
  use konstanten
  implicit none
  write (*,*) 'PI * E = ', PI * E
end subroutine calcPiMalE
! Ausgabe:
!   PI =    3.141593
!   E =    2.718282
!   Kreisflaeche fuer r=2.1 =    13.85442
!   PI * E =    8.539735
```

33.1.4. Datenabstraktion: Zusammenfassung von Daten und Methoden in einem Modul

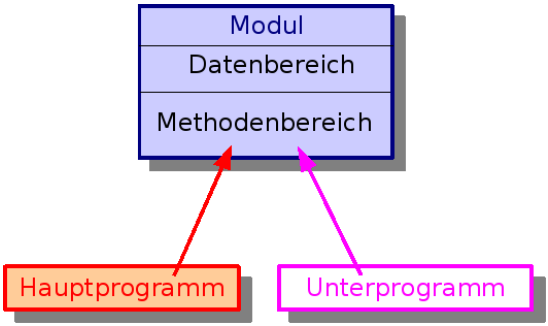


Abb. 39

Sind Daten und Unterprogramme als zusammengehörend zu betrachten, so ist es sinnvoll diese auch gemeinsam in einem Modul abzulegen. Ein Vorteil dabei ist, dass somit die Zugriffssteuerung gezielt eingesetzt werden kann. Moduldaten, die nicht von außerhalb des Moduls geändert werden dürfen, werden als `private` deklariert. Der Zugriff auf diese Daten kann dann nur noch mittels Methoden des Moduls erfolgen.

Beispiel: **Fortran 90/95-Code (free source form)**

```
module kreis
  implicit none
  save
  real, parameter :: PI = 3.141593
  real, private   :: r  = 0.0

  contains
    subroutine setR(val)
      real, intent(in) :: val
      r = val
    end subroutine setR
```

```
    real function getR()
      getR = r
    end function getR
    real function kreisflaeche()
      kreisflaeche = r**2 * PI
    end function kreisflaeche
end module kreis


! Hauptprogramm
program bsp
  use kreis
  implicit none

  call setR(5.0)
  write (*,*) "r = ", getR()
  write (*,*) "Flaeche = ", kreisflaeche()
  call sub1(5.0)
end program bsp


! Unterprogramm
subroutine sub1(val)
  use kreis
  implicit none
  real, intent(in) :: val

  call setR(val*2.56)
  write (*,*) "r = ", getR()
  write (*,*) "Flaeche = ", kreisflaeche()
end subroutine sub1
! Ausgabe:
!   r =  5.
!   Flaeche =  78.539825
!   r =  12.799999
!   Flaeche =  514.7185
```

33.1.5. Modul und Datenverbund

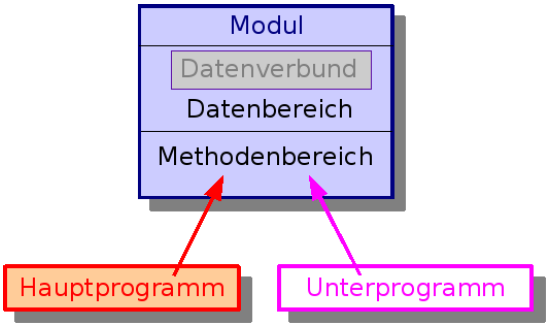


Abb. 40

Das Schlüsselwort `private` darf in einem Datenverbund nur in Verbindung mit einem Modul Anwendung finden. Einerseits kann die Sichtbarkeit aller Variablen im Datenverbund eingeschränkt werden. Anderseits kann auch der Datenverbund selbst als `private` deklariert werden. Ein Zugriff auf derart deklarierte Datenelemente ist dann nur noch durch Unterprogramme des gleichen Moduls möglich. Standardmäßig sind sowohl Datenverbund als auch seine Einzelkomponenten, so wie alle anderen nicht zugriffsbeschränkten Datenelemente, in einem Modul öffentlich (public).

Beispiel: Öffentlicher Datenverbund **Fortran 90/95-Code (free source form)**

```
module mod1
  save
  type :: tripel
    real:: x, y, z
  end type tripel
end module mod1
program bsp
  use mod1
```

```
  implicit none

  type(tripel) :: tr = tripel(10.5, 0.0, -6.5)

  write(*, *) tr
! Ausgabe:
!   10.5   0.0   -6.5
end program bsp
```

Beispiel: Öffentlicher Datenverbund mit privaten Datenelementen **Fortran 90/95-Code (free source form)**

```
module mod1
  save
  type :: tripel
    private
    real:: x, y, z
  end type tripel


  contains
    subroutine createTripel(this, valX, valY, valZ)
      type(tripel)     :: this
      real, intent(in) :: valX, valY, valZ

      this%x = valX
      this%y = valY
      this%z = valZ
    end subroutine createTripel


    subroutine writeTripel(this)
      type(tripel) :: this

      write(*,*) this
    end subroutine writeTripel
end module mod1
program bsp
  use mod1
  implicit none

  type(tripel) :: tr

  call createTripel(tr, 10.5, 0.0, -6.5)
  call writeTripel(tr)
! Ausgabe:
!   10.50000       0.000000      -6.500000
```

```
end program bsp
```

Beispiel: Privater Datenverbund

Fortran 90/95-Code (free source form)

```
module mod1
  save
  type, private :: tripel
    real :: x = 0.0, y=0.0, z=0.0
  end type tripel

  type(tripel), private :: t

  contains
    subroutine changeTripel(valX, valY, valZ)
      real, intent(in) :: valX, valY, valZ
      t = tripel(valX, valY, valZ)
    end subroutine changeTripel
    subroutine writeTripel()
      write (*,*) t
    end subroutine writeTripel
end module mod1
program bsp
  use mod1
  implicit none
! Hier könnte z.B. keine Variable vom Typ "triple" angelegt werden, da
 in dieser PE nicht
! sichtbar (privater Datenverbund des Moduls mod1)
  call writeTripel
! Ausgabe:
!    0.0 0.0 0.0
  call changeTripel(10.5, -5.0, -3.5)
  call writeTripel
! Ausgabe:
!    10.5 -5. -3.5
  call unterprogramm
! Ausgabe:
!    10.5 -5. -3.5
end program bsp
subroutine unterprogramm
  use mod1
  call writeTripel
end subroutine unterprogramm
```

Dieses Beispiel scheint auf den ersten Blick im Gegensatz zu den beiden vorherigen Varianten unnötige Einschränkungen aufzuweisen. Im Haupt- und Unterprogramm können keine Variablen des Typs `tripel` angelegt werden. Im Modul wird immer auf die gleiche Variable `t` zugegriffen. Das gewählte Beispiel könnte somit als SINGLETON[2] beschrieben werden. Es ist außerhalb des Moduls sichergestellt, dass immer nur ein Tripel existiert. Singletons werden auch in der objektorientierten Softwareentwicklung verwendet. Daneben sind auch andere Situationen vorstellbar, in denen sich das Konzept eines privaten Datenverbunds als nützlich erweisen kann.

33.1.6. Die Schnittstelle: Das Modul als Unterprogrammbibliothek

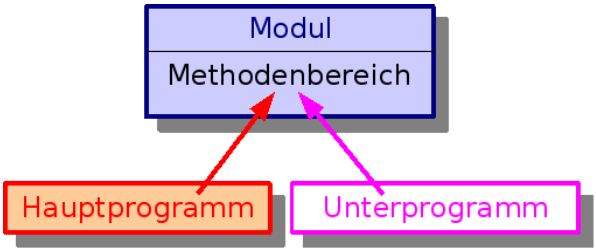


Abb. 41

2 HTTP://DE.WIKIPEDIA.ORG/WIKI/SINGLETON%20%28MATHEMATIK%29

Ein Modul kann als Sammelstelle für Unterprogramme dienen - quasi eine Bibliothek für häufig und in verschiedenen Kontexten benötigte Prozeduren.

Vorteile:

- Ordnung
- Modul kann in eine eigenständige Datei ausgelagert werden.
- Parameter-Datentypüberprüfung.

Beispiel: Parameter-Datentypüberprüfung

ohne Modul	**mit Modul**
program bsp implicit none integer :: zahl = 5 call unterprogramm(zahl) end program bsp subroutine unterprogramm(a) implicit none real, intent(in) :: a write (*,*) a end subroutine unterprogramm	module test contains subroutine unterprogramm(a) implicit none real, intent(in) :: a write (*,*) a end subroutine unterprogramm end module test program bsp use test implicit none integer :: zahl = 5 call unterprogramm(zahl) end program bsp

ohne Modul	mit Modul
• *gfortran* und *ifort* compilieren dieses Programm ohne Warnhinweis. • Der *g95* liefert in diesem Fall zumindest eine Warnung, compiliert jedoch auch das Programm. Die Datenausgabe zur Programmlaufzeit liefert jeweils einen fehlerhaften Wert.	Bereits beim Compilieren tritt eine Fehlermeldung auf, z.B. bei *g95*: In file ....f90:20 call unterprogramm(zahl) 1 Error: Type mismatch in parameter 'a' at (1). Passing INTEGER(4) to REAL(4) Das Programm lässt sich so nicht compilieren.

Der Schnittstellenblock: `interface`

Fortran 90/95 kennt das Sprachkonstrukt `interface`, welches sich in vielerlei Hinsicht nutzbringend verwenden läßt. Schnittstellenblöcke sind modulunabhängig verwendbar.

interface [*name*] [*interface-Spezifikationsteil*] end interface [*name*]

Der *interface-Spezifikationsteil* beinhaltet nur diejenigen Informationen, die für die Deklaration der Schnittstelle relevant sind (z.B. Unterprogrammbezeichnung, Variablendeklaration). Die genaue Festlegung der Unterprogramme (inkl. Ausführungsteil) erfolgt dann außerhalb des Schnittstellenblocks. Ein wesentliches Merkmal des Schnittstellenblocks ist die penible Überprüfung der Unterprogrammparameterdatentypen, so wie das auch beim Einsatz von Modulen geschieht.

Beispiel: Parameter-Datentypüberprüfung

ohne Schnittstellenblock	**mit Schnittstellenblock**
program bsp implicit none integer :: zahl = 5 call unterprogramm(zahl) end program bsp subroutine unterprogramm(a) implicit none real, intent(in) :: a write (*,*) a end subroutine unterprogramm	program bsp implicit none integer :: zahl = 5 interface subroutine unterprogramm(a) real, intent(in) :: a end subroutine unterprogramm end interface call unterprogramm(zahl) end program bsp subroutine unterprogramm(a) implicit none real, intent(in) :: a write (*,*) a end subroutine unterprogramm
• *gfortran* und *ifort* compilieren dieses Programm ohne Warnhinweis. • Der *g95* liefert in diesem Fall zumindest eine Warnung, compiliert jedoch auch das Programm. Die Datenausgabe zur Programmlaufzeit liefert jeweils einen fehlerhaften Wert.	Bereits beim Compilieren tritt eine Fehlermeldung auf, z.B. bei *gfortran*: In file ....f90:15 call unterprogramm(zahl) 1 Error: Type/rank mismatch in argument 'a' at (1) Das Programm lässt sich so nicht compilieren.

Generische Unterprogrammschnittstelle

Fortran 90/95 ermöglicht generische Methoden. Die im Schnittstellenblock deklarierten Unterprogramme können

damit über den gleichen Unterprogrammnamen aufgerufen werden. Intern erfolgt dann der Aufruf des jeweils passenden Unterprogramms durch die Unterschiede der Datentypen der Unterprogrammparameter. Mittel zum Zweck ist der benannte Schnittstellenblock.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  interface gensub
    subroutine writeReal(val)
      real, intent(in) ::val
    end subroutine writeReal
    subroutine writeInteger(val)
      integer, intent(in) ::val
    end subroutine writeInteger
    subroutine writeCharacter(val)
      character, intent(in) :: val
    end subroutine writeCharacter
  end interface gensub

  call gensub(5.5)
  call gensub(3)
  call gensub("H")
  call writeCharacter("X")
! Ausgabe:
!   Real-Wert =    5.500000
!   Integer-Wert =            3
!   Zeichen = H
!   Zeichen = X
end program bsp
subroutine writeReal(val)
  real, intent(in) ::val
  write (*,*) "Real-Wert = ", val
end subroutine writeReal
subroutine writeInteger(val)
  integer, intent(in) ::val
  write (*,*) "Integer-Wert = ", val
end subroutine writeInteger
subroutine writeCharacter(val)
  character, intent(in) ::val
  write (*,*) "Zeichen = ", val
end subroutine writeCharacter
```

Operatorüberladung
Fortran 90/95 verwendet von Haus aus Operatorüberladung. So können z.B. Variablen arithmetischen Datentyps einfach miteinander addiert werden:

```
c = a + b
```

Dieser Ausdruck funktioniert unabhängig davon, ob die Variablen vom Typ Ganzzahl oder Gleitkommzahl sind. Und auch für Felder funktioniert diese einfache Form der Addition. Die einzelnen Feldkomponenten werden korrekt addiert. Das ist in anderen Programmiersprachen nicht selbstverständlich. Zusätzlich kann in Fortran 90/95 auch der Programmierer mit Hilfe von Schnittstellenblöcken sogenannte *defined operations* festlegen.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
  interface operator (.PLUS.)
    function charAdd(c1, c2)
      character, intent(in) :: c1, c2
      character(len=2)      :: charAdd
    end function charAdd
  end interface operator (.PLUS.)
! "Addition" mittels definiertem .PLUS.-Operator
  write (*,*) "c1 .PLUS. c2 = ", "A" .PLUS. "B"
! oder auch mittels Funktion
  write (*,*) "charAdd () = ", charAdd("A", "B")
! Ausgabe:
!    c1 .PLUS. c2 = AB
!    charAdd () = AB
end program bsp
function charAdd(c1, c2)
  implicit none
  character, intent(in) :: c1, c2
  character(len=2)      :: charAdd
  charAdd = c1 // c2
end function charAdd
```

Beispiel: nur mit *gfortran* compilierbar, nicht mit *g95* oder *ifort*
Fortran 90/95-Code (free source form)

```
module mod
  type :: tripel
    real x, y, z
  end type
end module mod
program bsp
  use mod
  implicit none

  interface operator (*)
    function tripelMult(t1, t2)
      type(tripel), intent(in) :: t1, t2
      type(tripel)             :: tripelMult
    end function tripelMult
  end interface operator (*)
! Multiplikation mittels überladenem *-Operator
  write (*,*) "t1 * t2 =", tripel(2.0, 3.0, 4.0) * tripel(1.5, 0.5,
 2.0)
! oder auch mittels Funktion
  write (*,*) "tripelMult () =", tripelMult(tripel(2.0, 3.0, 4.0),
 tripel(1.5, 0.5, 2.0))
! Ausgabe:
!    t1 * t2 =   3.000000       1.500000       8.000000
!    tripelMult () =   3.000000       1.500000       8.000000
end program bsp
function tripelMult(t1, t2)
  use mod
  implicit none
  type(tripel), intent(in) :: t1, t2
  type(tripel)             :: tripelMult

  tripelMult = tripel(t1%x * t2%x, t1%y*t2%y, t1%z*t2%z)
end function tripelMult
```

Bei der Verwendung von *defined operations* sind jedoch einige Bedingungen zu beachten.

Neben den *defined operations* (`interface operator`) gibt es auch noch *defined assignments* (`interface assignment`) für die Überladung des Zuweisungsoperators.

Schnittstellenblock und Modul

Beim vorigen Beispiel bestand das Problem, dass nur einer der getesteten Compiler eine ausführbare Datei zu Stande brachte. Abhilfe schaffen kann die Verlagerung des Schnittstellenblocks in das Modul unter Zuhilfenahme von `module procedure`.

Beispiel: Diesen Programmcode schlucken sowohl *gfortran*, *g95* als auch *ifort* problemlos **Fortran 90/95-Code (free source form)**

```
module mod
  type :: tripel
    real x, y, z
  end type

  interface operator (*)
    module procedure tripelMult
  end interface operator (*)

  contains
    function tripelMult(t1, t2)
      implicit none
      type(tripel), intent(in) :: t1, t2
      type(tripel)             :: tripelMult

      tripelMult = tripel(t1%x * t2%x, t1%y*t2%y, t1%z*t2%z)
    end function tripelMult
end module mod
program bsp
  use mod
  implicit none
! Multiplikation mittels überladenem *-Operator
  write (*,*) "t1 * t2 =", tripel(2.0, 3.0, 4.0) * tripel(1.5, 0.5,
 2.0)
! oder auch mittels Funktion
  write (*,*) "tripelMult () =", tripelMult(tripel(2.0, 3.0, 4.0),
 tripel(1.5, 0.5, 2.0))
! Ausgabe:
!    t1 * t2 =   3.000000       1.500000       8.000000
!    tripelMult () =   3.000000       1.500000       8.000000
end program bsp
```

Unterprogramme: Die Übergabe optionaler und benannter Parameter

Der Einsatz optionaler Parameter wurde bereits im Kapitel UNTERPROGRAMME[3] abgehandelt. Zusätzlich unterstützt Fortran 90/95 auch die Verwendung von benannten Parametern (argument keywords). Dazu ist es erforderlich, dass das Unterprogramm ein explizites Interface aufweist, wie das z.B. bei Einbindung von Unterprogrammen in Module automatisch der Fall ist.

Beispiel: **Fortran 90/95-Code (free source form)**

```
module m1
  contains
    subroutine abc( var1, var2 )
      implicit none

      integer, intent( in ) :: var1, var2

      write( *, * ) var1, var2
    end subroutine abc
end module m1
program bsp
  use m1
  implicit none
  call abc( 12, 99 )
  call abc( var2 = 99, var1 = 12 )
  call abc( var2 = 12, var1 = 99 )
  call abc( 12, var2 = 99 )
! call abc( var1 = 12, 99 )    ! so funktioniert das nicht
! Ausgabe:
!     12          99
!     12          99
!     99          12
!     12          99
end program bsp
```

3 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20FORTRAN%2095%3A%20UNTERPROGRAMME%23OPTIONALE%20PARAMETER

33.1.7. Sonstiges

- `use` *modulbezeichnung*, `only :` *modulelemente*
- `use` *modulbezeichnung* `=>` *bezeichnung*

33.2. Objektorientierte Programmierung

Modulares Programmieren reißt heutzutage niemanden mehr vom Hocker und selbst objektorientierte Konzepte sind seit spätestens Ende der 80-Jahre des vergangenen Jahrhunderts Stand der Technik. Im akademischen Bereich waren diese Dinge natürlich schon früher bekannt. Fortran hat in diesem Bereich also nie eine Vorreiterrolle inne gehabt. Nach jeweils kurzen Nachdenkphasen haben aber die Fortran-Leute unerschrocken auch in diesem Bereich nachgezogen.

Fortran bildet also aufgrund seiner langen Geschichte ein weites Spektrum der Programmierparadigmen ab: vom prozeduralen Programmieren (FORTRAN 77) über das modulare Programmieren (Fortran 90/95) hin zum objektorientierten Programmieren (Fortran 2003).

Fortran 90/95 ist keine typisch objektorientierte Sprache. Trotzdem lässt sich mit den modularen Spracheigenschaften von Fortran 90/95 bereits in weiten Bereichen Objektorientierung simulieren beziehungsweise nachbauen. Aber erst mit Fortran 2003 lässt sich wirklich komfortabel objektorientiert programmieren. Diese Fortran-Version wirbt dann auch „offiziell“ mit diesem Merkmal.

Welche Eigenschaften verbindet man typischerweise mit Objektorientierung? Da wären

1. Klassen - Datenkapselung und Datenabstraktion
2. Objekte

3. Vererbung
4. Statische Polymorphie: Überladen von Funktionen (und Operatoren)
5. Run-Time-Polymorphie

Welche dieser Merkmale bietet Fortran 90/95 von Haus aus und was läßt sich relativ einfach nachbauen?

33.2.1. Klassen - Datenkapselung und Datenabstraktion

Was ist eine Klasse?

- KLASSE (OBJEKTORIENTIERTE PROGRAMMIERUNG)[4]
- KLASSE (UML)[5]

Ein Fortran 90/95-Modul sieht sehr ähnlich aus wie eine Klasse in der objektorientierten Programmierung und ist dementsprechend auch Ausgangspunkt für den objektorientierten Ansatz in Fortran 90/95.

Klasse	**Fortran 90/95**
Klassenbezeichner Datenbereich (Attribute) Methodenbereich (Operationen)	module ... ! Datenbereich contains ! Methodenbereich end module ...

Einen grundlegenden Mechanismus zum Schreiben objektorientierter Programme stellt Fortran 90/95 somit in Form der Module zur Verfügung. Jetzt stellt sich aber die Frage, wie diese Klasse instantiiert werden soll. Das wird im Abschnitt *Objekte* gezeigt.

4 HTTP://DE.WIKIPEDIA.ORG/WIKI/KLASSE%20%28OBJEKTORIENTIERTE%20PROGRAMMIERUNG%29

5 HTTP://DE.WIKIPEDIA.ORG/WIKI/KLASSE%20%28UML%29

33.2.2. Objekte

Was sind Objekte? OBJEKT (PROGRAMMIERUNG)[6]

Ein einfaches Modul mit ein paar konventionellen Datenelementen und Prozeduren ist noch keine Klasse, aus der Objekte zu erzeugen wären. Dazu bedarf es noch eines Unterscheidungskriteriums zwischen den einzelnen Objekten. Dieses Unterscheidungskriterium kann, wie auch schon früher im Teilkapitel *Modul und Datenverbund*[7] gezeigt, mittels Datenverbund hergestellt werden. Die Objekte werden somit nicht über den Modulnamen als Klasse angesprochen, sondern über die Datenverbund-Bezeichnung. Auch das ist keine neue Erkenntnis, sondern eine geschickte Ausnutzung eines Mechanismus, den Fortran 90/95 standardmäßig mit Modulen und Datenverbund zur Verfügung stellt.

Beispiel:

6 HTTP://DE.WIKIPEDIA.ORG/WIKI/OBJEKT%20%28PROGRAMMIERUNG%29

7 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20FORTRAN%2095%3A%20MODULE%20UND%20OOP%23MODUL%20UND%20DATENVERBUND

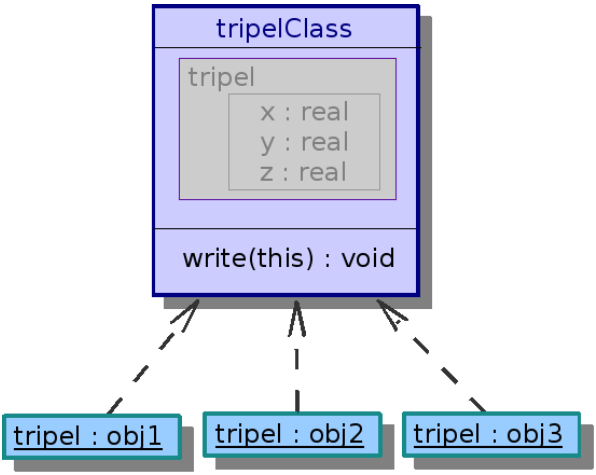


Abb. 42

Fortran 90/95-Code (free source form)

```
module tripelClass
  implicit none
  save
  type :: tripel
    real:: x, y, z
  end type tripel

  contains
    subroutine write(this)
      type(tripel), intent(in) :: this

      write(*, *) "********** Tripel **********"
      write(*, *) this
    end subroutine

end module tripelClass
program bsp
  use tripelClass
  implicit none

  type(tripel) :: obj1 = tripel(1.5, 0.0, -6.5)
```

```
  type(tripel) :: obj2 = tripel(2.5, 1.0, -4.5)
  type(tripel) :: obj3 = tripel(3.5, 2.0, -2.5)

  call write(obj1)
  call write(obj2)
  call write(obj3)

! Ausgabe:
!   ********** Tripel **********
!      1.500000       0.000000      -6.500000
! ********** Tripel **********
!      2.500000       1.000000      -4.500000
! ********** Tripel **********
!      3.500000       2.000000      -2.500000
end program bsp
```

Weitere wichtige Elemente beim Erzeugen und Zerstören von Objekten sind in der OOP die sogenannten Konstruktoren und Destruktoren.

Was sind Konstruktoren und Destruktoren? KONSTRUKTOREN UND DESTRUKTOREN[8]

Konstruktoren und Destruktoren kennt Fortran 90/95 natürlich nicht. Im Bedarfsfall sind diese Elemente also mittels konventioneller Unterprogramme nachzubilden und dann jeweils explizit in den entsprechenden Programmabschnitten, nach Erzeugung beziehungsweise beim Abbau des jeweiligen Objektes, manuell aufzurufen.

33.2.3. Vererbung

Was ist Vererbung? VERERBUNG (PROGRAMMIERUNG)[9]

8 HTTP://DE.WIKIPEDIA.ORG/WIKI/KONSTRUKTOREN%20UND%20DESTRUKTOREN

9 HTTP://DE.WIKIPEDIA.ORG/WIKI/VERERBUNG%20%28PROGRAMMIERUNG%29

Das OOP-Prinzip „Vererbung“ ist in Fortran 90/95 nur über Umwege realisierbar. Fortran 90/95 kennt konzeptionsbedingt für diesen Zweck keinen einfachen programmtechnischen Mechanismus. Nachfolgend wird anhand eines kleinen und überschaubaren Beispiels eine mögliche Lösung demonstriert.

Beispiel:

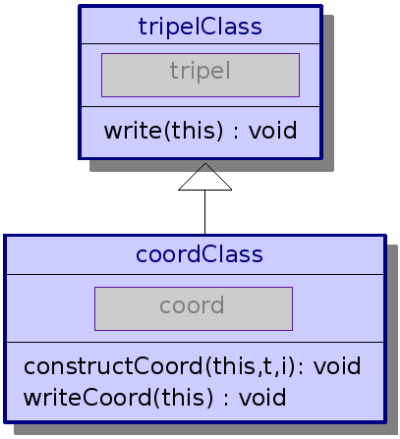


Abb. 43

Fortran 90/95-Code (free source form)

```
module tripelClass
  implicit none
  save
  type :: tripel
    real:: x, y, z
  end type tripel

  contains
    subroutine write(this)
      type(tripel), intent(in) :: this

      write(*, *) "********** Tripel **********"
      write(*, *) this
```

```
    end subroutine
end module tripelClass
module coordClass
  use tripelClass
  implicit none
  save

  type :: coord
    type(tripel) :: tr
    integer      :: id
  end type

  contains
    subroutine constructCoord(this, t, i)
      implicit none

      type(coord), intent(out) :: this
      type(tripel), intent(in) :: t
      integer                  :: i

      this%tr = t
      this%id = i
    end subroutine constructCoord


    subroutine writeCoord(this)
      implicit none

      type(coord), intent(in) :: this
      write(*, "(A, I5, A)") "****************** KOORDINATE ",
 this%id, " ******************"
      call write(this%tr)
    end subroutine writeCoord
end module coordClass
program bsp
  use coordClass
  implicit none

  type(coord) :: obj1 = coord(tripel(1.5, 0.0, -6.5), 1005)
  type(coord) :: obj2 = coord(tripel(2.5, 1.0, -4.5), 1006)
  type(coord) :: obj3 = coord(tripel(3.5, 2.0, -2.5), 1007)

  call writeCoord(obj1)
  call writeCoord(obj2)
  call writeCoord(obj3)

! Ausgabe:
!   ****************** KOORDINATE  1005 ******************
```

```
!    ********** Tripel **********
!      1.500000       0.000000      -6.500000
!   ****************** KOORDINATE  1006 ******************
!    ********** Tripel **********
!      2.500000       1.000000      -4.500000
!   ****************** KOORDINATE  1007 ******************
!    ********** Tripel **********
!      3.500000       2.000000      -2.500000
end program bsp
```

33.2.4. Statische Polymorphie: Überladen von Funktionen (und Operatoren)

Das Überladen von Funktionen und Operatoren unterstützt Fortran 90/95 wiederum standardmäßig. Diese Konzepte wurden schon in den Abschnitten

- GENERISCHE_UNTERPROGRAMMSCHNITTSTELLE[10]
- OPERATORÜBERLADUNG[11] und
- SCHNITTSTELLENBLOCK UND MODUL[12]

kurz erläutert.

33.2.5. Run-Time-Polymorphie

Polymorphie zur Laufzeit eines Programmes ist mit den Mitteln von Fortran 90/95 nur einigermaßen aufwendig und kompliziert

10 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN_95%3A_MODULE_UND_OOP%23GENERISCHE_UNTERPROGRAMMSCHNITTSTELLE

11 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN_95%3A_MODULE_UND_OOP%23OPERATOR.C3.BCBERLADUNG

12 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN_95%3A_MODULE_UND_OOP%23SCHNITTSTELLENBLOCK%20UND%20MODUL

nachzubauen, jedoch prinzipiell möglich. Es sei zu diesem Thema auf die weiterführende Literatur verwiesen.

33.3. Ausblick

Sind die Methoden zur Nachbildung objektorientierter Mechanismen auch in Fortan 2003 verwendbar? Prinzipiell schon, jedoch bietet Fortran 2003 bessere und erweiterte Möglichkeiten zur Behandlung dieses Themas. Dort wurde nämlich das Prinzip des Datenverbunds wesentlich erweitert. Ein `type`-Block kann dann neben den Datenelementen auch Unterprogramme beinhalten, das Schlüsselwort `extends` zwecks Vererbungmechanismus ist vorhanden, etc. Das in Fortran 90/95 als Datenbund/Struktur verwendbare `type`-Konstrukt wurde also in Fortran 2003 zu einer Klasse, ähnlich wie es in anderen Programmierspachen unter der Bezeichnung `class` bekannt ist, aufgerüstet.

33.4. Literatur

33.4.1. Quellen

- DECYK: SCIENTIFIC COMPUTING WITH FORTRAN 95[13]
- GRAY, ROBERTS: OBJECT-BASED PROGRAMMING IN FORTRAN 90[14]

13 HTTP://EXODUS.PHYSICS.UCLA.EDU/FORTRAN95/PSTIRESEARCHLECSERIES1.HTML

14 HTTP://WWW.CCS.LANL.GOV/CCS/CCS-{}4/PDF/OBF90.PDF

33.4.2. Weiterführende Literatur

- Decyk, Norton, Szymanski: Introduction to Object-Oriented Concepts Using Fortran90, 1996[15]
- Decyk, Norton, Szymanski: How to Express C++ Concepts in Fortran90. 1997[16]
- Decyk, Norton, Szymanski: How to Support Inheritance and Run-Time Polymorphism in Fortran90, 1998[17]
- Decyk, Norton: A Simplified Method for Implementing Run-Time Polymorphism in Fortran95, 2004[18]
- Decyk, Gardner: Object-Oriented Design Patterns in Fortran95, 2006[19]

15 http://exodus.physics.ucla.edu/Fortran95/F90_Objects.pdf

16 http://exodus.physics.ucla.edu/Fortran95/ExpressC++.pdf

17 http://exodus.physics.ucla.edu/Fortran95/Stopwatch.pdf

18 http://exodus.physics.ucla.edu/Fortran95/simplifiedRT.pdf

19 http://exodus.physics.ucla.edu/Fortran95/FortranPatterns.pdf

34. Offizielle Fortran 95-Erweiterungen

Hier sind jene ISO/IEC-Standards angeführt, die nicht Teil des Original-Fortran 95-Standards sind, sondern erst später als offizielle Fortran 95-Erweiterungen formuliert wurden. Etliche Fortran 95-Compiler bieten diese Features derzeit noch nicht oder nur teilweise.

34.1. Zeichenketten variabler Länge

- Offizielle Bezeichnung: Fortran, Part 2, Varying length character strings
- Standards:
  ftp://ftp.liv.ac.uk/pub/fortran_std/is1539-2.html ISO/IEC 1539-2 : 1994

  ftp://ftp.nag.co.uk/sc22wg5/N1351-N1400/N1375.pdf ISO/IEC 1539-2 : 2000

- Referenzmodul-Download-Adressen:
  ftp://ftp.nag.co.uk/sc22wg5/ISO_VARYING_STRING/Sample_Module/iso_vs
  - programmiert in Standard-Fortran 95 und daher portabel

  ftp://ftp.nag.co.uk/sc22wg5/ISO_VARYING_STRING/Sample_Module/iso_vs
  - verwendet Sprachmittel gem. Fortran 95-Erweiterung ISO/IEC TR 15581 : 1999 bzw. 2001 (enhanced data type facilities). Dieses Modul soll effizienter als iso_vst.f95 sein und Speicherlecks vermeiden

Diese Fortran 95-Erweiterung ermöglicht in einfacher Weise das Arbeiten mit Zeichenketten variabler Länge. Die notwendigen Datenelemente und Funktionen sind im Modul `iso_varying_string` hinterlegt. Die nötigen Programmabläufe zur dynamische Anpassung des jeweils erforderlichen Speicherplatzes übernehmen somit die Unterprogramme dieses Moduls. Eine explizite Begrenzung der maximalen Zeichenkettenlänge ist nicht vorgesehen. Einschränkungen ergeben sich nur durch Hardwarerestriktionen bzw. die Programmkomplexität.

In den getesteten aktuellen Fortran 95-Compilern (gfortran, g95 und ifort) ist mit Stand 01.01.2007 diese Funktionalität noch nicht inkludiert. Zur Nutzung dieses Features mit diesen Compilern ist deshalb der Download und die Einbindung einer der o.a. Moduldateien erforderlich (oder man schreibt selbst ein entsprechendes standardkonformes Modul).

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  use iso_varying_string
  implicit none

  type( varying_string ) :: str
  str = "Wiki"
  write (*,*) len(str)
  write (*,*) char(str)
  str = str // "books"
  write (*,*) len(str)
  write (*,*) char(str)
! Ausgabe
!   4
!   Wiki
!   9
!   Wikibooks
end program bsp
```

Compilieren, Linken:

```
gfortran -o bsp iso_vsta.f95 bsp.f90
```

Prinzipiell gelten für Zeichenketten mit variabler Länge die gleichen generisch-intrinsischen Funktionsbezeichner wie für normale Zeichenketten. Auch Stringkonkatenation, Zuweisung und Vergleichsoperationen sind wie gewohnt erlaubt. Zuätzlich kommen noch einige neue Funktionen hinzu:

Funktionsbezeichner	**Kommentar**
var_str	Datentypkonvertierung. `character` → `varying_string`
get, put, put_line	I/O
insert	Einfügen eines Teilstrings
replace	Ersetzen eines Teilstrings
remove	Entfernen eines Teilstrings
extract	Extrahierung eines Teilstrings
split	Einen String in zwei Strings splitten.

34.2. Bedingte Compilierung

- Standard: [ftp://ftp.nag.co.uk/sc22wg5/N1301-N1350/N1306.pdf ISO/IEC 1539-3 : 1998. Conditional Compilation]

Mannigfaltig sind die Gründe, welche den Einsatz bedingter Compilierung (conditional compilation, Kurzform: coco) erstrebenswert erscheinen lassen. Seien es die unterschiedlichen Betriebssystemkonventionen zur Vergabe von Pfad- und Dateinamen, unterschiedliche Anforderungen an Entwickler- und Releaseversionen eines Programmes oder auch spezielle interna-

tionale Marktbedürfnisse, die Programmvarianten erfordern. All diese Bedürfnisse und noch viel mehr kann coco befriedigen.

Weder g95, noch gfortran oder ifort bieten bisher von Haus aus die Möglichkeit zur "conditional compilation". Es gibt aber externe Tools zur ISO/IEC 1539-3, die als Präprozessor fungieren können und somit das gewünschte Verhalten erzeugen, z.B. das Programm COCO[1].

Beispiel:

Die Datei *bsp.fpp:*

```
?? integer, parameter :: varianteA = 1, varianteB = 2
?? integer :: flag = 1
program bsp
?? if ( flag == varianteA ) then
   write (*,*) "Variante A"
?? else if (flag == varianteB) then
   write (*,*) "Variante B"
?? endif
end program bsp
```

mit der Set-Datei *bsp.set*:

```
?? integer :: flag = 2
?? alter: delete
```

ergibt nach einem Präprozessorlauf `coco bsp` folgendes Fortran-Programm bsp.f90

```
program bsp
   write (*,*) "Variante B"
```

1 HTTP://USERS.EROLS.COM/DNAGLE/COCO.HTML

```
end program bsp
```

Beispiel:

Wird die coco-Anweisung `alter: delete` in der Set-Datei weggelassen, so werden die überflüssigen Zeilen nicht gelöscht, sondern nur auskommentiert (shift). Standardmäßig entspricht dies beim coco-Präprozessor einer `alter: shift3`-Set-Anweisung (!?>). Die Set-Datei bsp.set

```
?? integer :: flag = 2
```

würde mit der obigen bsp.fpp-Datei nach einem `coco bsp` also diesen Fortran-Code liefern

```
!?>?? integer, parameter :: varianteA = 1, varianteB = 2
!?>?? integer :: flag = 1
program bsp
!?>?? if ( flag == varianteA ) then
!?>   write (*,*) "Variante A"
!?>?? else if (flag == varianteB) then
   write (*,*) "Variante B"
!?>?? endif
end program bsp
!?>?? This was produced using the following SET file
!?>?? integer :: flag = 2
```

34.3. Floating-point Exceptions

- Standard: [ftp://ftp.nag.co.uk/sc22wg5/N1351-N1400/N1378.pdf ISO/IEC TR 15580 : 1998. Floating-point exception handling]

Im Fortran 2003-Standard sind Module betreffend IEEE 754-Standard (IEEE Standard for Binary Floating-Point Arithmetic) enthalten. Näheres dazu wird im Fortran 2003-Kapitelabschnitt DIE INTRINSISCHEN IEEE-MODULE[2] beschrieben.

34.4. Allocatable Components

- Standard: [ftp://ftp.nag.co.uk/sc22wg5/N1351-N1400/N1379.pdf ISO/IEC TR 15581 : 1999. Enhanced data type facilities]

Diese Erweiterung bezieht sich auf das Schlüsselwort `allocatable`. In Standard-Fortran 90/95 ist die dynamische Allokation von Speicherplatz für Felder mit dem Attribut `allocatable` eigentlich nur in einem lokalen Kontext möglich. Der *Technical Report 15581* fordert, dass solche Felder darüber hinaus auch in den Anwendungsbereichen

- Rückgabewert von Funktionen
- Unterprogrammparameter
- Feldelemente in Datenverbunden

uneingeschränkt verwendbar sein sollen.

Die *Enhanced Data Type Facilities* werden bereits standardmäßig vom aktuellen *g95-*, *gfortran-*, *ifort-* und *Sun-Fortran*-Compiler unterstützt.

Beispiel: Rückgabewert **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none
```

2 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20FORTRAN%202003%3A%20INTRINSISCHE%20MODULE%23DIE%20INTRINSISCHEN%20IEEE-{}MODULE

```
  write ( *, * ) fun_all( 5 )
  write ( *, * ) fun_all( 7 )
! Ausgabe:
!   1 2 3 4 5
!   1 2 3 4 5 6 7
  contains
    function fun_all( n )
      implicit none

      integer, dimension( : ), allocatable :: fun_all
      integer, intent( in )                :: n
      integer                              :: j, st

      allocate( fun_all( n ), stat = st )
      if( st == 0 ) then
        forall( j = 1 : n )
          fun_all(j) = j
        end forall
      else
        write( *, * ) "Allocate-Fehler"
      end if
  end function fun_all
end program bsp
```

Beispiel: Unterprogrammparameter **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  integer, dimension( : ), allocatable :: dynarr
  integer                              :: j

  allocate( dynarr( 3 ) )
  forall( j = 1 : 3 )
    dynarr( j ) = j * 2
  end forall
  call fun_all( dynarr )

  write( *, * ) "Out: ", dynarr
  deallocate( dynarr )
  allocate( dynarr( 5 ) )

  forall( j = 1 : 5 )
    dynarr( j ) = j * 3
  end forall
```

```
  call fun_all( dynarr )
  deallocate( dynarr )
! Ausgabe:
!   Argument: 2 4 6
!   (De)Allocate im UP: 88 99
!   Out:  88 99
!   Argument: 3 6 9 12 15
!   (De)Allocate im UP: 88 99
  contains
    subroutine fun_all( a )
      implicit none

      integer, dimension( : ), allocatable, intent( inout ) :: a
      write( *, * ) "Argument:", a

      deallocate( a )
      allocate( a (2) )
      a(1) = 88
      a(2) = 99
      write( *, * ) "(De)Allocate im UP:",  a
   end subroutine fun_all
end program bsp
```

Mit Standard-Fortran 90/95 könnten sie zwar das allozierte Feld als Parameter aus dem Hauptprogramm an das Unterprogramm übergeben, dort wäre es aber nicht als `allocatable` kennzeichenbar und somit im Unterprogramm nicht in der gezeigten Art und Weise (de)allozierbar. Mittels Standardkonformitäts-Compileroptionen, z.B.

- *Intel Fortran Compiler:* `-stand`
- *g95*, *gfortran*: `-std=f95`

ist überprüfbar, welche Möglichkeiten Standard-Fortran 95 bietet und welche Eigenschaften den Erweiterungen zuschreibbar sind.

Beispiel: *allocatable array* im Datenverbund **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  type struktur
```

```
    integer                                :: nr
    integer, dimension( : ), allocatable :: arr
  end type struktur
  type( struktur ) :: a1, a2

  allocate( a1%arr( 5 ) )
  allocate( a2%arr( 2 ) )

  a1%nr     = 9453
  a1%arr(1) = 1
  a1%arr(5) = 5
  a2%nr     = 9454
  a2%arr(1) = 11
  a2%arr(2) = 22

  write ( *, * ) "a1 =" , a1%nr, a1%arr
  write ( *, * ) "a2 =",  a2%nr, a2%arr
! Ausgabe:
!   a1 = 9453 1 0 0 0 5
!   a2 = 9454 11 22
end program bsp
```

Teil V.

Fortran 2003

35. Programmaufbau

35.1. Programmaufbau und Zeilenformat

Der grundlegende Programmaufbau und das Zeilenformat von *Fortran 90/95* wurden in *Fortran 2003* beibehalten. Neben der *free source form* ist aus Kompatibilitätsgründen auch noch immer die alte, aus *FORTRAN 77* bekannte, *fixed source form* gültig.

Eine Zeile darf auch in *Fortran 2003* bei Verwendung der *free source form* standardmäßig maximal 132 Zeichen beinhalten. Ein symbolischer Name darf nun höchstens 63 Zeichen lang sein. Eine Anweisung darf sich maximal über 256 Zeilen erstrecken. Das Zeilenfortsetzungszeichen ist wie in *Fortran 90/95* das Kaufmanns-Und: `&`.

35.2. Zeichenvorrat

Der *Fortran 2003*-Zeichenvorrat wurde gegenüber *Fortran 90/95* erweitert:

35.3. Anwendungsgebiet der neu hinzugekommenen Zeichen

Von den in *Fortran 2003* neu zugefügten Zeichen haben nur die eckigen Klammern einen konkreten Anwendungsbereich als Kennzeichnung von Feldkonstruktoren. Ein Feldkonstruktor darf neben der aus *Fortran 95* bekannten Form

```
(/ werte /)
```

nun auch mit eckigen Klammern geschrieben werden

```
[ werte ]
```

Beispiel: **Fortran 2003-Code**

```
! bsp.f03
program bsp
  implicit none

  integer, dimension( 3 ) :: a = [ 20, 33, 55 ]
  integer, dimension( 2 ) :: b = (/ 44, 55 /)

  write( *, *) a
  write( *, *) b
! Ausgabe:
!   20 33 55
!   44 55
end program bsp
```

Kompilieren, linken:

```
g95 -o bsp bsp.f03
```

36. Datentypen

36.1. Intrinsische Datentypen

Die aus Fortran 90/95 bekannten intrinsischen Datentypen sind weiterhin uneingeschränkt gültig.

Datentyp	**Kommentar**
integer	Ganzzahlen
real	Reelle Zahlen einfacher Genauigkeit
(double precision)	Reelle Zahlen doppelter Genauigkeit
complex	Komplexe Zahlen
logical	Logischer Datentyp (.true., .false.)
character	Zeichen(kette)

Im Umfeld des `character`-Datentyps gab es einige kleinere Ergänzungen:

- Unterstützung internationaler Zeichensätze in Rahmen der ISO 10646-Norm (UNIVERSAL CHARACTER SET[1])

1 HTTP://DE.WIKIPEDIA.ORG/WIKI/UNIVERSAL%20CHARACTER%20SET

- Die Funktion `selected_char_kind` ist neu. Ihre Funktionsweise ist äquivalent zu den bereits aus Fortran 90/95 bekannten Funktionen `selected_int_kind` und `selected_real_kind`.

36.2. Enumeration

Mit Fortran 2003 sind auch in dieser Programmiersprache Enumerationen (Aufzählungstypen) möglich. Die Werte in einer solchen Enumeration besitzen einen `integer`-Datentyp.

enum, bind(C) enumerator :: *wert(e)* // ... end enum

Die von C-Enumerationen bekannten Eigenschaften gelten gleichermaßen für Fortran-Enumerationen, z.B.:

- ohne explizite Zuweisung von Werten wird mit dem Wert 0 gestartet.
- ohne explizite Zuweisung von Werten wird in der Anordnungsreihenfolge der Elemente sukzessiv immer um 1 hochgezählt.
- Wurde dem Vorgängerelement eine Ganzzahl zugewiesen, dem Element jedoch nicht, so ist der Wert dieses Elementes die dem Vorgängerelement zugeordnete Ganzzahl + 1.

Beispiel: **Fortran 2003-Code**

```
program bsp
  implicit none
  integer :: wert
  enum, bind( C )
    enumerator :: MON, DIE, MIT = 3, DON, FRE = 5, SAM = 66, SON = 77
  end enum

  write( *, * ) MON
  write( *, * ) MIT
  write( *, * ) SAM
```

```
  wert = 4

  if( wert == MIT + 1 ) then
    write( *, * ) "Donnerstag"
  endif
! Ausgabe:
!   0
!   3
!   66
!   Donnerstag
end program bsp
```

Obwohl der Fortran 2003-Working Draft J3/04-007 durch das `bind(c)` eine zwingende Anbindung an einen Aufzählungstyp in der Programmiersprache C vorgaukelt, ist dies nicht der Fall. Fortran-Enumerationen funktionierten auch ohne entsprechendes C-Gegenstück. Sie werden im Fortran 2003-Working Draft auch nicht im Abschnitt *Interoperability with C* geführt, sondern direkt im Abschnitt über die Fortran-Datentypen. Diese Tatsache dürften die Compilerbauer auch unterschiedlich interpretieren. So wäre beim *g95*-Compiler das Attribut `bind(C)` nicht unbedingt erforderlich, ist jedoch möglich. Beim Einsatz des *gfortran* ist der Zusatz `bind(C)` obligatorisch. Aus Kompatibilitätsgründen ist somit immer die strikte Schreibweise nach Fortran 2003-Standard empfehlenswert. Der *Intel Fortran Compiler 9.1* unterstützt dieses Sprachmerkmal ohnehin noch nicht.

36.3. Derived Type

Der *Derived Type* wurde in Fortran 2003 wesentlich erweitert. Er ist nun kein reiner Datenverbund wie noch in Fortran 90/95, sondern eine richtige Klasse. Genauer beschrieben wird dieser Typ später im Kapitel zur OOP, hier zunächst nur einige einfache Erweiterungsmerkmale gegenüber Fortran 90/95.

36.3.1. Benannte Parameter

Beispiel:

Fortran 2003-Code

```
program bsp
  implicit none
  type test
    integer        :: i
    character( 20) :: str
  end type


  type( test ) :: t1 = test ( i = 1, str = "Wiesenfeld" )

  write( *, *) t1
! Ausgabe:
!   1 Wiesenfeld
end program bsp
```

Die Verwendung von benannten Parametern bei der Konstruktion einer *Derived Type*-Variable kann vor allem in Verbindung mit Vorgabewerten sinnvoll sein.

36.3.2. Vorgabewerte

Ab *Fortran 2003* dürfen Variablen in einem *Derived Type* auch mit Vorgabewerten (default values) belegt werden.

Beispiel: **Fortran 2003-Code**

```
program bsp
  implicit none
  type test
    integer        :: i   = -1
    character( 20) :: str = "NN"
  end type

  type( test ) :: t1 = test ()
  type( test ) :: t2 = test ( str = "Wiesenfeld" )
  type( test ) :: t3 = test ( str = "Walddorf", i = 1 )
```

```
  write( *, *) t1
  write( *, *) t2
  write( *, *) t3
! Ausgabe:
!   -1 NN
!   -1 Wiesenfeld
!   1 Walddorf
end program bsp
```

37. Zeiger

37.1. Prozedurenzeiger

37.1.1. Einführung

Mit dem 2003er-Standard unterstützt auch Fortran Zeiger auf Prozeduren (procedure pointer, Funktionszeiger). Diese spielen auch eine wichtige interne Rolle bei der Realisierung objektorientierter Mechanismen und beim C-Binding. Schematisch wird ein Prozedurenzeiger so deklariert:

procedure([*name*]), pointer :: *zeigername*

Die Angabe von `name` ist optional. Wird an dieser Stelle ein Name, z.B. ein Unterprogrammbezeichner angegeben, so bedeutet dies, dass der Prozedurenzeiger mit allen Unterprogrammen, die das gleiche Interface aufweisen, kompatibel ist. Prozedurenzeiger können wie normale Zeiger gehandhabt werden.

Beispiel: **Fortran 2003-Code**

```
program bsp
  implicit none
  procedure(up), pointer :: pptr => null()

  pptr => ooops

  call pptr

! Ausgabe:
```

```
!   ooops

  contains
    subroutine ooops()
      write( *, * ) "ooops"
    end subroutine ooops
    subroutine up()
    end subroutine up
end program bsp
```

37.1.2. Prozedurenzeiger mit implizitem Interface

Bei der Deklaration eines Prozedurenzeigers muss keine explizite Schnittstelle angegeben werden. Im Folgenden wird dies am Beispiel eines mit `pptr2` benannten Prozedurenzeigers demonstriert.

Beispiel: **Fortran 2003-Code**

```
program bsp
  implicit none
  procedure( up ) , pointer :: pptr1 => null()
  procedure( )    , pointer :: pptr2 => null()
  procedure( add ), pointer :: pptr3 => null()

  pptr1 => ooops
  pptr2 => ooops
  call pptr1
  call pptr2

! Ausgabe:
!   ooops
!   ooops
  pptr3 => add
  write( *, * ) pptr3( 5 , 12 )
! Ausgabe:
!   17
! Folgende auskommentierte Zuordnung waere nicht erlaubt:
! pptr1 => add


  contains
    subroutine up()
```

```
    end subroutine up
    subroutine ooops()
      write( *, * ) "ooops"
    end subroutine ooops


    function add( a, b )
      integer               :: add
      integer, intent( in ) :: a, b

      add = a + b
    end function add
end program bsp
```

37.1.3. Abstraktes Interface

Das bei der Deklaration eines Prozedurenzeigers als Schnittstelle genannte Unterprogramm muss nicht real implementiert sein. Es können statt dessen auch abstrakte Interfaces Verwendung finden. Diese sind gleich wie konventionelle Interface-Blöcke aufgebaut, mit dem Unterschied, dass sie als `abstract interface` gekennzeichnet sind. Ein mit einem abstrakten Interface deklarierter Prozedurzeiger passt dann für jedes Unterprogramm, welches mit identer Schnittstelle ausgestattet ist.

Fortran 2003-Code

```
program bsp
  implicit none
  abstract interface
    function afunc( x, y )
      integer               :: afunc
      integer, intent( in ) :: x, y
    end function afunc
  end interface


  procedure( afunc ), pointer :: pptr1 => null()
  procedure( add )  , pointer :: pptr2 => null()

  pptr1 => add
```

```
  write( *, * ) pptr1( 5 , 12 )
! Ausgabe: 17

  pptr1 => mult
  write( *, * ) pptr1( 3 , 2 )
! Ausgabe: 6
! Folgendes funktioniert uebrigens auch, da add() und mult() das
 gleiche Interface
! aufweisen:
  pptr2 => mult
  write( *, * ) pptr2( 5 , 5 )
! Ausgabe: 25


  contains
    function add( a, b )
      integer               :: add
      integer, intent( in ) :: a, b

      add = a + b
    end function add
    function mult( a, b )
      integer               :: mult
      integer, intent( in ) :: a, b

      mult = a * b
    end function mult
end program bsp
```

37.2. Zeiger und das `intent`-Attribut

Nun ist bei der Übergabe von Zeigern an Unterprogramme auch die Angabe eines `intent`-Attributs möglich. Das war mit *Fortran 90/95* noch nicht erlaubt. Diese `intent`-Angaben beziehen sich aber nicht auf die Variablenwerte an sich, sondern beschränken nur die Möglichkeiten zur Zeigerzuordnung im Unterprogramm selbst.

Beispiel: **Fortran 2003-Code**

```
program bsp
```

```
  implicit none
  integer, target  :: x = 15
  integer, pointer :: ptr1 => null(), ptr2 => null()
  ptr1 => x
  ptr2 => x

  call mult( ptr1, ptr2)

  write( *, *) "Zuordnungsstatus ptr1:", associated( ptr1 )
  write( *, *) "Zuordnungsstatus ptr2:", associated( ptr2 )
  write( *, *) "Wert ptr1:", ptr1
  write( *, *) "Wert x:", x
! Ausgabe:
!   Zuordnungsstatus ptr1: T
!   Zuordnungsstatus ptr2: F
!   Wert ptr1: 45
!   Wert x: 45


  contains
    subroutine mult( a, b )
      integer, pointer, intent( in )    :: a
      integer, pointer, intent( inout ) :: b
      integer, target :: val = 3

! Folgendes waere nun nicht erlaubt, da a nur intent( in )
!      a => null()
! Das auch nicht:
!       a => val
! Das allerdings ist erlaubt:
       a = a * val
! b ist mit intent( inout ) spezifiziert, also ist hier eine
 Zeigerzuordnung
! erlaubt:
      b => null()

    end subroutine mult
end program bsp
```

37.3. Zeiger und Felder

Auch im Zusammenspiel von Zeigern mit Feldern bringt der *Fortran 2003*-Standard einige Ergänzungen.

38. Ein- und Ausgabe

38.1. Streams

Fortran 2003 bietet zusätzlich zum altbekannten datensatzbasierten I/O nun auch Dateieingabe und -ausgabe in Form von Streams, wie das z.B. in der Programmiersprache C seit jeher üblich ist.

38.1.1. Unterschied zum alten I/O-Konzept?

- Streams ermöglichen das Lesen und Schreiben von Binärdateien, ohne sich mit den auf Datensätzen aufbauenden Strukturen der konventionellen Fortran-I/O herumschlagen zu müssen.
- Bei der Ein-/Ausgabe mit Streams wird die Datei als kontinuierliche Byte-Sequenz betrachtet.
- Stream-I/O ist logischerweise nicht für interne Dateien gedacht.

38.1.2. Anwendung

Auch wenn Stream-I/O in *Fortran 2003* ein neues Konzept ist, so sind die altbekannten `open`-, `read`-, `write`-und `close`-Befehle dafür zuständig.

- Öffnen eines Streams:

```
open( ..., access = "STREAM", ... )
```

- Lesen und Schreiben:

```
read( ... ) ... write( ... ) ...
```

- Schließen eines Streams:

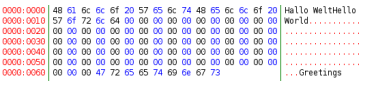
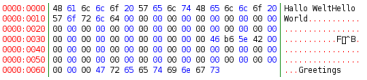
```
close( ... )
```

38.1.3. Unformatierte Stream-I/O

Beispiel: **Fortran 2003-Code**

```
program bsp
  implicit none
  real                 :: a = 55.678
  real                 :: b
  character( len = 3 ) :: str
  open( 50, file = "test", access = "STREAM", status = "REPLACE")

  write( 50 ) "Hallo Welt"
  write( 50 ) "Hello World"
! Ausgabe in Datei: siehe Bild 1

  write( 50, pos = 100 ) "Greetings"
! Ausgabe in Datei: siehe Bild 2

  write( 50, pos = 60 ) a
! Ausgabe in Datei: siehe Bild 3
  read( 50, pos = 60 ) b
  write( *, * ) b
! Ausgabe:
!   55.678
  read( 50, pos = 8 ) str
  write( *, * ) str
! Ausgabe:
!   elt
  close( 50 )
end program bsp
```

Da es sich um eine unformatierte Ein-/Ausgabe handelt, darf natürlich kein Formatspezifizierer bei den `read`- und `write`-Anweisungen angegeben werden, auch kein `*`. Mittels `pos`-Spezifizierer kann an eine bestimmte Position in der Datei gesprungen werden.

Bild 1	 Abb. 44
Bild 2	 Abb. 45
Bild 3	 Abb. 46

38.1.4. Formatierte Stream-I/O

Beispiel: **Fortran 2003-Code**

```
program bsp
  implicit none
  real                    :: a = 55.678
  real                    :: b = 13.9876
  character( len = 20 ) :: str1, str2, str3
  integer                 :: fposition
  open( 50, file = "test", access = "STREAM", form = "FORMATTED",
 status = "REPLACE")

  write( 50, "(2A20)" ) "Hallo Welt", "Hello World"
  inquire( 50, pos = fposition )
  write( *, * ) fposition
  write( 50, * ) a, new_line( "x" ), b, " abcdef"
  read( 50, *, pos = fposition ) a
  write( *, * ) a
```

```
  read( 50, *, pos = 1 ) str1, str2, str3
  write( *, * ) str1, str2, str3

  read( 50, * ) str1, a
  write( *, * ) str1, a
! Ausgabe in Datei:
!              Hallo Welt         Hello World
!   55.678
!   13.9876  abcdef
! Ausgabe auf Bildschirm:
!   42
!   55.678
!   Hallo               Welt               Hello
!   55.678               13.9876

  close( 50 )
end program bsp
```

38.2. Asynchrone I/O

38.3. Rekursive I/O

38.4. Sonstiges

38.5. Weblinks

- STREAM INPUT/OUTPUT IN FORTRAN[1]

1 HTTP://WWW.STAR.LE.AC.UK/%7ECGP/STREAMIO.HTML

39. Intrinsische Funktionen und Subroutinen

Die *Fortran 90/95*-Funktionen und -Subroutinen sind natürlich auch in *Fortran 2003* uneingeschränkt gültig. Einige Unterprogramme wurden neu aufgenommen, andere in ihrer Funktionalität etwas erweitert.

39.1. Neu

39.1.1. Datentypfunktionen

Funktion	Beschreibung
i = **selected_-char_kind** (c)	Gibt den kind-Wert des Parameters zurück. Der Rückgabewert ist von Datentyp `integer`. Parameter: • name Rückgabewert: • `name = "DEFAULT"`: der Default-Wert für Zeichen wird zurückgegeben • `name = "ASCII"`: der Wert für ASCII-Zeichen wird zurückgegeben • `name = "ISO_10646"`: der Wert für "ISO 10646"-Zeichen wird zurückgegeben • -1 ... Zeichentyp wird nicht unterstützt Beispiel: i = selected_char_kind("ASCII") i => 1

39.1.2. Kommandozeile und Environment

Funktion	Beschreibung
i = **command_-argument_-count** ( )	Anzahl der übergebenen Kommandozeilenargumente (der Programmname selbst wird nicht mitgezählt). Der Rückgabewert ist vom Typ `integer`. Beispiel: Programmaufruf mit: ./a.out opt1 opt2 i = command_argument_count( ) i => 2

Subroutine	**Beschreibung**

get_command ([c, i, i])	Übergebene Kommandozeilenargumente (ohne Programmname)
	Parameter:
	• command: Kommandozeilenargumente als Zeichenkette, `intent( out )`, optional
	• length: Länge der Zeichenkette, `intent( out )`, optional
	• status: Status, `intent( out )`, optional
	  • 0 ... OK
	  • -1 ... command-Argument existiert und ist kürzer als length
	  • andere Zahl ... Fehler
	Beispiel: Programmaufruf mit: ./a.out opt1 opt2
	call get_command(str, len, st)
	str => opt1 opt2 len => 9 st => 0

Subroutine	**Beschreibung**

get_command_argument (i, [c, i, i])	Ein bestimmtes Kommandozeilenargument (inkl. Programmname)
	Parameter:
	• number: Nummer des gewünschten Kommandozeilenargumentes beginnend bei 0, `intent( in )`
	• value: Wert des Kommandozeilenargumentes, `intent( out )`, optional
	• length: Länge der Zeichenkette, `intent( out )`, optional
	• status: Status, `intent( out )`, optional
	  • 0 ... OK
	  • -1 ... Argument existiert und ist kürzer als length
	  • andere Zahl ... Fehler
	Beispiel: Programmaufruf mit: ./a.out opt1 opt2
	call get_command_argument(1, str, len, st)
	str => opt1 len => 4 st => 0

Subroutine	Beschreibung

get_environment_variable (c1, [c2, i, i, l])	Wert einer bestimmten Umgebungsvariable.
	Parameter:
	• name: Name der Umgebungsvariable, `intent( in )`
	• value: Wert der Umgebungsvariablen, `intent( out )`, optional
	• length: Länge der Zeichenkette `c2`, `intent( out )`, optional
	• status: Status, `intent( out )`, optional
	• trim_name: `intent( in )`, optional
	  • 0 ... OK
	  • -1 ... Umgebungsvariable existiert, Wert ist kürzer als length
	  • 1 ... Umgebungsvariable existiert nicht
	  • 2 ... keine Unterstützung von Umgebungsvariablen
	  • andere Zahl ... sonstiger Fehler
	Beispiel:
	call get_environment_variable("PWD", str, len, st, .TRUE.)
	str => /usr/bin len => 8 st => 0

39.2. Erweitert

- `system_clock(i1, ir, i2)` ... Das zweite Argument (`count_rate`) darf nun vom Datentyp `integer` oder `real` sein.
- `max, maxloc, maxval, min, minloc, minval` ... Funktionieren nunmehr auch für Werte vom Datentyp `character`.
- `atan2(r1, r2), log(rx), sqrt(rx)` ... Unterscheidung von positiven und negativen Nullen im Argument.

40. Intrinsische Module

40.1. Grundlegendes

Module gab es bereits mit *Fortran 90/95*. Neu in *Fortran 2003* sind die sogenannten *"intrinsischen Module"*. Das sind jene Module, die bereits standardmäßig von Fortran-2003-Compilern bereitgestellt werden. Werden Datenelemente oder Funktionen aus solchen *intrinsischen Modulen* benötigt, so ist das entsprechende Modul mittels

```
use, intrinsic :: modulname
```

in die jeweilige Programmeinheit einzubinden.

Der Unterschied zu konventionellen (nonintrinsischen) Modulen ist das Wörtchen `intrinsic`, das dem Compiler mitteilt, dass er das Modul bereits mitbringt und nicht irgendwo extern danach suchen soll. Wird nach dem `use`-Schlüsselwort kein entsprechendes Attribut oder das `non_intrinsic`-Attribut angegeben, so zeigt dies an, dass ein nonintrinsisches Modul benutzt wird.

40.2. Das intrinsische Modul `iso_fortran_env`

Das `iso_fortran_env`-Modul enthält einige Fortran-umgebungsspezifische Konstanten.

Beispiel:

Fortran 2003-Code

```
program bsp
  use, intrinsic :: iso_fortran_env
  implicit none
  write( *, * ) INPUT_UNIT
  write( *, * ) OUTPUT_UNIT
  write( *, * ) ERROR_UNIT
  write( *, * ) IOSTAT_END
  write( *, * ) IOSTAT_EOR
  write( *, * ) NUMERIC_STORAGE_SIZE
  write( *, * ) CHARACTER_STORAGE_SIZE
  write( *, * ) FILE_STORAGE_SIZE
! Ausgabe, z.B.:
!   5
!   6
!   0
!   -1
!   -2
!   32
!   8
!   8
end program bsp
```

Erläuterung:

Konstante	**Anmerkung**
INPUT_UNIT	Standard-Eingabeeinheit (entspricht `unit=*` bei `read`)
OUTPUT_UNIT	Standard-Ausgabeeinheit (entspricht `unit=*` bei `write`)
ERROR_UNIT	Standard-Fehlerausgabeeinheit
IOSTAT_END	end-of-file (EOF)
IOSTAT_EOR	end-of-record (EOR)
NUMERIC_STORAGE_SIZE	Speicherplatzbedarf (in bits)

Konstante	Anmerkung
CHARACTER_STORAGE_-SIZE	Speicherplatzbedarf (in bits)
FILE_STORAGE_SIZE	Speicherplatzbedarf (in bits)

All diese Konstanten sind Skalare vom Datentyp `integer`.

40.3. Das intrinsische Modul `iso_c_binding`

Das `iso_c_binding`-Modul liefert die Konstanten und Unterprogramme, die für die Einbindung von C-Bibliotheken in Fortran-Programme erforderlich sind. Näheres dazu findet sich im Kapitel FORTRAN 2003 UND C[1].

40.4. Die intrinsischen IEEE-Module

Die bereits aus dem dem TR 15580 : 1998 (floating-point exception handling) bekannten Module

- `ieee_exceptions`
- `ieee_arithmetic`
- `ieee_features`

wurden in *Fortran 2003* in Form von intrinsischen Modulen aufgenommen. Diese Module decken den IEEE 754-1985-Standard (auch IEC 559:1989) ab.

1 Kapitel 61.5 auf Seite 478

40.4.1. Wovon handelt der IEEE 754-1985-Standard?

Im Bereich der Gleitkommazahlen (`real`, `float`, ...) herrschte bis in die 1980er-Jahre Anarchie. Es gab keine verbindlichen Regeln wie Gleitkommazahlen repräsentiert werden, wie gerundet wird, wie Under- und Overflows gehandhabt werden, wie mit Unendlich und NaN verfahren wird, etc. Das führte dazu, dass das gleiche Computerprogramm auf unterschiedlichen Rechnerarchitekturen und mit verschiedenen Compilern unterschiedliche Resultate liefern konnte. Um diesem Manko zu begegnen wurde in den frühen 1980er-Jahren eine Standardisierung angestrebt. Resultat war die Verabschiedung des IEEE 754-1985-Standards (IEEE Standard for Binary Floating-Point Arithmetic for microprocessor systems).

Dieser Standard regelt im Wesentlichen

- die Repräsentation von Gleitkommazahlen:

Darstellung: $zahl = (-1)^s \cdot m \cdot b^e$

s	...	Vorzeichen
m	...	Mantisse
b	...	Basis (2 für normalisierte Zahlen)
e	...	Exponent

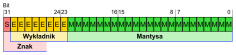


Abb. 47

Zahlenformate:

single	...	4 Bytes
double	...	8 Bytes

double-extended	...	≥10 Bytes, optional

- die Darstellung normalisierter und denormalisierter Zahlen:
  - Normalisierte Zahlen: GLEITKOMMAZAHLEN#NORMALISIERUNG[2]
  - Denormalisierte Zahlen: Bereich zwischen der kleinsten darstellbaren normalisierten Zahl und Null (Exponent hat einen reservierten Wert, führende Bit der Mantisse ist 0)
- NaN (Not a Number), $\pm\infty$, ± 0
- Rundungen (zur nächstgelegenen darstellbaren Zahl, in Richtung Null, in Richtung $+\infty$ oder in Richtung $-\infty$)
- das Verhalten verschiedener Operationen (Grundrechenarten, Wurzelberechnung, Konvertierung Gleitkommazahl → Ganzzahl, Binär-Dezimal-Konvertierung, Vergleiche mit Nan und ∞, etc.)
- Exception-Handling (Overflow, Underflow, Division by Zero, Inexact, Invalid)

Weiterführende Weblinks:

- IEEE 754[3]
- Kahan, W.: Why do we need a floating-point arithmetic standard?, UC Berkeley, 1981, HTTP://HTTP.CS.BERKELEY.EDU/ WKAHAN/IEEE754STATUS/WHY-IEEE.PDF[4]
- Kahan, W.: Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic, UC

2 HTTP://DE.WIKIPEDIA.ORG/WIKI/GLEITKOMMAZAHLEN%23NORMALISIERUNG

3 HTTP://DE.WIKIPEDIA.ORG/WIKI/IEEE%20754

4 HTTP://HTTP.CS.BERKELEY.EDU/~{}WKAHAN/IEEE754STATUS/WHY-{}IEEE.PDF

Berkeley, 1996, HTTP://HTTP.CS.BERKELEY.EDU/ WKAHAN/IEEE754STATUS/IEEE754.PDF[5]

- LINKSAMMLUNG BEI "CENTRE CHARLES HERMITE"[6]
- IEEE 754-1985 "STANDARD FOR BINARY FLOATING-POINT ARITHMETIC"[7]

40.4.2. Implementierung in Fortran 2003

Wie bereits erwähnt, besitzt Fortran 2003 intrinsische Module, mit denen der Zugriff auf bestimmte IEEE-Eigenschaften erfolgen kann. Dazu stehen eine Reihe von Funktionen, Subroutinen, Verbundtypen und Konstanten zur Verfügung. Merkmal ist, dass diese immer mit dem Präfix `ieee` beginnen.

`ieee_arithmetic`

Das aus Programmiersicht umfangreichste Modul ist sicherlich `ieee_arithmetic`. Dieses enthält zahlreiche Funktionen, Subroutinen und Konstanten.

Abfragefunktionen für die Unterstützung bestimmter IEEE-Elemente, z.B.:

---

5 HTTP://HTTP.CS.BERKELEY.EDU/~{}WKAHAN/IEEE754STATUS/IEEE754.PDF

6 HTTP://CCH.LORIA.FR/DOCUMENTATION/IEEE754

7 HTTP://754R.UCBTEST.ORG/STANDARDS/754.PDF

l = ieee_support_datatype([x])	Prüft ob die IEEE-Arithmetik für einen speziellen `real`-Datentyps, charakterisiert durch `x` (Zahl oder Feld), unterstützt wird. Wird kein Argument angegeben, so wird geprüft, ob die IEEE-Arithmetik für alle `real`-Datentypen unterstützt wird.
l = ieee_support_nan([x])	Prüft, ob NaN-Werte unterstützt werden
l = ieee_support_rounding(round_value, [x])	Prüft, ob ein bestimmter IEEE-Rundungsmodus unterstützt wird. Mögliche Rundungsmodi sind: • `ieee_nearest` • `ieee_to_zero` • `ieee_up` • `ieee_down` • `ieee_other`

Elementare Funktionen, z.B.:

l = ieee_is_finite(x)	Prüft, ob der Wert `x` endlich ist
r = ieee_next_after(x, y)	Liefert die nächste darstellbare Zahl von `x` in Richtung `y`

r = ieee_rint(x)	Rundet gemäß eingestelltem Rundungsmodus zu einer Ganzzahl und liefert diese Zahl mit dem Datentyp von `x` zurück.

Die Kind-Funktion:

r = ieee_selected_real_kind([p, r])	Liefert einen kind-Wert

Nichtelementare Subroutinen, z.B.:

ieee_get_underflow_mode(gradual)	Liefert den aktuellen Underflow-Modus
ieee_set_rounding_mode(round_value)	Setzt den IEEE-Rundungsmodus, mögliche Werte für `round_value` • `ieee_nearest` • `ieee_to_zero` • `ieee_up` • `ieee_down` • `ieee_other`

ieee_exceptions

Das ieee_exceptions-Modul enthält zwei Funktionen, mit denen abgefragt werden kann, welche Exceptions unterstützt werden bzw. inwieweit IEEE-Halting unterstützt wird:

```
l = ieee_support_flag( flag, [x] )
l = ieee_support_halting( flag )
```

Mögliche Flags sind

- ieee_invalid
- ieee_overflow
- ieee_divide_by_zero
- ieee_underflow
- ieee_inexact

Desweiteren sind in diesem Modul einige Subroutinen zum Setzen bzw. Abfragen diverser Flags enthalten:

```
ieee_get_status( status_value )
ieee_set_flag( flag, flag_value )
ieee_set_halting_mode( flag, halting )
ieee_set_status( status_value )
```

Bei Einbindung des ieee_arithmetic-Moduls ist auch automatisch Zugriff auf die public-Elemente des Moduls ieee_exceptions gegeben.

ieee_features

Das ieee_features-Modul liefert einige benannte Konstanten, z.B. ieee_datatype, ieee_inf, ieee_sqrt.

Beispiel: Rundungsmodus

IEEE-Subroutinen:

```
ieee_get_rounding_mode( val )
ieee_set_rounding_mode( flag )
```

Mögliche Wert für `flag` sind:

- `ieee_nearest` ... default, Rundung zur nächstgelegenen Zahl (wenn das nicht eindeutig möglich ist, dann Rundung zur nächstgelegenen geraden Zahl)
- `ieee_to_zero` ... Rundung in Richtung 0
- `ieee_down` ... Rundung Richtung $-\infty$
- `ieee_up` ... Rundung Richtung $+\infty$

Fortran 2003-Code

```
program bsp
  use, intrinsic :: ieee_arithmetic
  implicit none
```

```
  real, dimension(6)    :: a = (/ -1.5, -0.5, 0.5, 1.5, 2.5, 3.5 /)

! Standard-Fortran-Rundungsfunktion
  write( *, * ) anint( a )
! IEEE-Rundungsfunktion (default)
  write( *, * ) ieee_rint( a )

! IEEE-Rundungsfunktion mit Flag ieee_round_type = ieee_nearest
  call ieee_set_rounding_mode( ieee_nearest )
  write( *, * ) ieee_rint( a )

! IEEE-Rundungsfunktion mit Flag ieee_round_type = ieee_to_zero
  call ieee_set_rounding_mode( ieee_to_zero )
  write( *, * ) ieee_rint( a )
! IEEE-Rundungsfunktion mit Flag ieee_round_type = ieee_down
  call ieee_set_rounding_mode( ieee_down )
  write( *, * ) ieee_rint( a )
! IEEE-Rundungsfunktion mit Flag ieee_round_type = ieee_up
  call ieee_set_rounding_mode( ieee_up )
  write( *, * ) ieee_rint( a )
end program bsp
```

Ausgabe:

Standard-Fortran	**IEEE-Default**	**ieee_-nearest**	**ieee_-to_zero**	**ieee_-down**	**ieee_-up**
-2.0 -1.0 1.0 2.0 3.0 4.0	-2.0 0.0 0.0 2.0 2.0 4.0	-2.0 0.0 0.0 2.0 2.0 4.0	-1.0 0.0 0.0 1.0 2.0 3.0	-2.0 -1.0 0.0 1.0 2.0 3.0	-1.0 0.0 1.0 2.0 3.0 4.0

Beispiel: Halting-Modus

Dieser Modus bestimmt, ob nach einer Exception das Programm angehalten oder fortgesetzt wird. Die in diesem Beispiel eingesetzten IEEE-Unterprogramme sind:

- `ieee_support_halting( flag )` ... prüft, ob auf dem System das IEEE-Halting für das angegebenen Flag überhaupt unterstützt wird (Rückgabewert: `.true.`). Mögliche Flags:
  - `ieee_invalid`
  - `ieee_overflow`
  - `ieee_divide_by_zero`
  - `ieee_underflow`
  - `ieee_inexact`
- `ieee_set_halting_mode( flag, mode )` ... setzt den Halting-Modus für ein bestimmtes Flag.
  - `flag` ... wie bei `ieee_support_halting( flag )`
  - `mode`:
    - `.true.` ... anhalten
    - `.false.` ... Programm weiter ausführen

Fortran 2003-Code

```
program main
  use, intrinsic :: ieee_arithmetic
  implicit none

  real :: a, b

  if( ieee_support_halting( ieee_divide_by_zero ) ) then
    call ieee_set_halting_mode( ieee_divide_by_zero, .false. )

    read( *, * ) a, b
    write( *, * ) "Resultat: ", a / b
    write( *, * ) "Programm wird fortgesetzt ..."
  else
    write( *, * ) "IEEE-Halting wird nicht unterstuetzt"
  end if
end program main
```

Eingabe:

- 10.0 (... für a)
- 0.0 (... für b)

Ausgabe (bei Programmerstellung mit dem Sun-Express-Fortran-Compiler f95):

`ieee_set_halting_mode( ieee_divide_by_zero, .false. )`	`ieee_set_halting_mode( ieee_divide_by_zero, .true. )`
Resultat: Inf Programm wird fortgesetzt ...	Gleitkomma-Ausnahme

Weitere Beispiele zum Thema "Exceptions and IEEE arithmetic" sind im Fortran 2003-Working Draft J3/04-007 ab Seite 386 enthalten.

Teil VI.

Bibliotheken

Eine Programmbibliothek bezeichnet in der Programmierung eine Sammlung von Programmfunktionen für zusammengehörende Aufgaben. Bibliotheken sind im Unterschied zu Programmen keine eigenständig lauffähigen Einheiten, sondern Hilfsmodule, die Programmen zur Verfügung gestellt werden.

40.5. Quelltextbibliotheken

Quelltextbibliotheken enthalten Sammlungen von Wertedefinitionen, Deklarationen, Funktionen, Klassen, generischen Bestandteilen, usw.

40.5.1. API

Eine Programmierschnittstelle ist eine Schnittstelle die von einem Softwaresystem anderen Programmen zur Anbindung an das System zur Verfügung gestellt wird. Oft wird dafür die Abkürzung *API* (für engl. *application programming interface*, deutsch: *Schnittstelle zur Anwendungsprogrammierung*) verwendet. Im Gegensatz zu einer Binärschnittstelle (ABI) definiert ein API nur die Verwendung der Schnittstellen auf Quelltextebene.

Neben dem Zugriff auf Datenbanken, die Hardware wie Festplatte oder Grafikkarte kann ein API auch das Erstellen von Komponenten der grafischen Benutzeroberfläche ermöglichen oder vereinfachen.

Im weiteren Sinne wird die Schnittstelle jeder Bibliothek (Library) als API bezeichnet.

40.6. Statische Bibliotheken

Statische Bibliotheken werden nach dem Kompiliervorgang durch einen so genannten Linker oder Binder in einem eigenen Schritt mit dem ausführbaren Programm verbunden.

Der Linker sucht aus den Bibliotheksdateien Unterprogramme heraus, für die es im Programm keine Implementierung gibt. Diese werden dann aus den Dateien extrahiert und an das Programm gebunden, d.h. der Unterprogrammcode wird an den Programmcode angefügt und die Aufrufverweise werden auf die Unterprogrammadressen gerichtet.

40.7. Dynamische Bibliotheken

Dynamische Bibliotheken werden erst bei Bedarf in den Arbeitsspeicher geladen und durch den sogenannten Lader mit dem ausführbaren Programm verbunden. Dadurch muss eine Bibliothek, die von mehreren Programmen genutzt wird, nur einmal im Speicher gehalten werden.

Dies ist beispielsweise bei Multitasking-Systemen vorteilhaft, wenn die Bibliotheken insgesamt sehr groß sind und von vielen Prozessen gleichzeitig verwendet werden. Dort wird eine Bibliotheksdatei bei ihrer ersten Verwendung in den Speicher geladen. Trifft ein Programm auf den Verweis zu einem Unterprogramm, das noch nicht eingebunden wurde, dann wird ein Laufzeitbinder aktiviert. Dieser sucht das Unterprogramm in den im Speicher vorhandenen Bibliotheken, fügt die Adresse am Aufrufpunkt ein und führt das Unterprogramm erstmalig aus.

Bei jedem weiteren Aufruf des Unterprogramms ist dann die Adresse vorhanden, so dass das Unterprogramm direkt aufgerufen

wird. Die Ausführungszeit, insbesondere die Startzeit eines Programms, ist hier geringfügig erhöht. Dies wird in Kauf genommen, da der Programmcode der Bibliotheksfunktionen von allen Prozessen geteilt wird. Der Speicherbedarf aller Programme zusammen ist daher in der Regel kleiner als beim statischen Linken.

Unterstützt das Betriebssystem virtuellen Speicher, so entfällt das Laden der gesamten Bibliothek bei der ersten Verwendung. Stattdessen wird die Bibliothek in den Speicherbereich jedes sie verwendenden Prozesses eingeblendet. Die virtuelle Speicherverwaltung lädt danach nur tatsächlich benötigte Teile der Bibliothek bei Bedarf von der Festplatte in den Arbeitsspeicher.

40.8. Bibliotheken in verschiedenen Programmiersprachen

Bibliotheken in Programmiersprachen enthalten Leistungen, die nicht im Compiler implementiert sind, sondern in der Sprache selbst programmiert sind und mit dem Compiler zusammen oder völlig von ihm getrennt dem Programmierer zur Verfügung stehen. Im ersten Fall ist die Bibliothek meist in der Sprachbeschreibung festgelegt. Im zweiten Fall spricht man von einer externen Bibliothek.

40.9. Bibliotheken bei verschiedenen Betriebssystemen

40.9.1. Windows

Bei den Betriebssystemen Windows und auch bei OS/2 wird eine Bibliotheksdatei, die dynamisch bindet, als Dynamic Link Library (DLL) bezeichnet. Entsprechend haben diese Dateien meist die Dateiendung *.dll*. Ihr Dateiformat ist *Portable Executable*.

Problematisch ist bei Windows 95, Windows 98 und Windows Me, dass durch unzureichende Schutzmaßnahmen die DLLs nicht kontrolliert werden - jedes Programm darf sie austauschen und kann dem Betriebssystem damit möglicherweise Schaden zufügen. Windows 2000 und Windows XP hingegen verfügen über einen Systemschutz, der auch die DLLs einbezieht.

Vorteile

- Außer Code können auch Daten (z. B. Dialog-Ressourcen) von mehreren Prozessen gemeinsam genutzt werden.
- DLLs werden häufig statisch gelinkt, können aber auch dynamisch (daher der Name) gelinkt werden. Dynamisch heißt hier, dass die DLL explizit vom Programm zur Laufzeit geladen wird und die Funktionen, die sich in der DLL befinden, „per Hand“ mit dem Programm verbunden werden. Dadurch wird es möglich, durch Austauschen der DLL die Funktionalität des Programms zur Laufzeit zu verändern.
- DLLs können unabhängig vom Hauptprogramm gewartet werden. D. h. Funktionen in der DLL können ohne Wissen des Programms verändert werden. Danach wird die DLL einfach ausgetauscht (die alte DLL-Datei wird überschrieben), ohne dass das Hauptprogramm verändert werden muss.

- Da die DLL als unabhängige Datei dem Hauptprogramm beiliegen muss, können Anbieter von Programmcode besser sicherstellen, dass Programmierer, die die Funktionen ihrer DLL nutzen, dafür auch bezahlen. Die Funktionalität der DLL verschwindet so nicht (wie bei einer Library) im Code des Programms. Dieser Vorteil wird von Befürwortern freier Software als Nachteil gesehen.

Nachteile

Änderungen in DLLs ziehen oft auch Änderungen im Programm mit sich. Dadurch kommt es leicht zu Versionskonflikten, die oft nur sehr schwer aufzuspüren sind.

Eine der Grundideen der DLLs war, Programmcode zwischen mehreren Programmen zu teilen, um so kostbaren Speicher zu sparen. In der Praxis ist es jedoch dazu gekommen, dass viele Programme bei der Installation DLLs in das Windows-Systemverzeichnis schreiben, die außer diesem speziellen Programm kein anderes benutzen kann.

Außerdem ist die Entwicklung und insbesondere die Anbindung im Vergleich aufwändiger als zur statischen Bibliothek.

Quintessenz

DLLs sollte man nur benutzen, wenn man ihre spezielle Funktionalität benötigt und man ausschließlich unter Windows arbeitet. Sind statische Bibliotheken für den Zweck ausreichend, sollte man diese vorziehen. In der Praxis ergeben sich keinerlei Einsparungen bei der Größe des Codes.

40.9.2. Unix-artige

Auf Unix-artigen Betriebssystemen ist für dynamische Bibliotheken die Bezeichnung *shared library* (englisch *shared*, geteilt) gebräuchlich.

Für diese Dateien hat sich die Endung .so (*s***'hared** o**bject***) eingebürgert. In der Regel folgt dem Bibliotheksnamen noch eine Versionsnummer.*

41. Grafik und GUI

41.1. DISLIN

DISLIN[1]

1 HTTP://DE.WIKIPEDIA.ORG/WIKI/%20DISLIN

42. Allgemeines

Die *DISLIN Scientific Plotting Software* ist eine Bibliothek für die grafische Datendarstellung. Auch für die Gestaltung grafischer Benutzeroberflächen läßt sich *DISLIN* verwenden. *DISLIN* greift zu diesem Zwecke auf die *Motif*-Bibliothek zu. Die *DISLIN*-Bibliothek ist für mehrere Programmiersprachen konzipiert, so auch auch für die Programmiersprache Fortran.

43. Beispiele

43.1. Beispiel 1: Strings und Zahlen

Fortran 90/95-Code (free source form)

```
program dbsp1
  implicit none

  real, parameter :: PI = 3.1415926

  ! *** Initialisierung ***
  call setpag ("DA4P")                ! DIN-A4 Hochformat
  call metafl ("CONS")                ! Ausgabe auf Konsole
 (Bildschirm)
  call disini                         ! DISLIN initialisieren

  ! *** Zeichnen ***
  call messag ("Hallo, Welt!", 50,50)  ! Message schreiben
  call number (PI, 4, 50, 150)         ! 3.1416 schreiben

  ! *** Aufräumen ***
  call disfin                         ! DISLIN beenden
end program dbsp1
```

Programm erstellen:

- Variante 1:

```
gfortran -c dateiname.f95
dlink dateiname
```

- Variante 2:

```
gfortran -o dateiname dateiname.f95 -ldislin
```

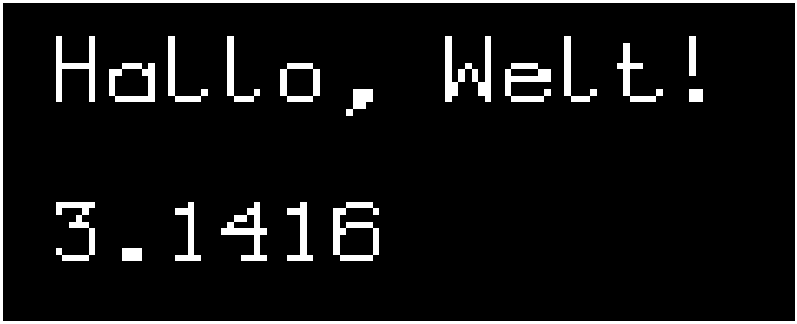


Abb. 48

Eine Auswahl von möglichen aktuellen Parametern für die Subroutine `metafl`:

- "CONS" ... Konsole (Bildschirm)
- "XWIN" ... X-Window (Bildschirm)
- "EPS" ... Encapsulated Postscript-Datei
- "PNG" ... PNG-Datei
- "SVG" ... SVG-Datei
- "PDF" ... PDF-Datei

43.2. Beispiel 2: Zeichnen von Kurven und Funktionen

Fortran 90/95-Code (free source form)

```
program dbsp2
  implicit none
```

```
  real, dimension(0:99) :: x, y
  integer :: i, setrgb

  do  i = 0, 99
    x(i) = i / 20.0
    y(i) = sin(x(i))
  end do

  ! *** Initialisierung ***
  call setpag ("DA4P")                                    ! DIN-A4
  call metafl ("PNG")                                     ! Ausgabe in
 eine PNG-Datei
  call disini                                             ! DISLIN
 initialisieren

  ! *** Zeichnen ***
  call pagfll (255)                                       !
 Hintergrundfarbe auf weiß setzen
  call color (setrgb (0., 0., 0.))                        !
 Vordergrundfarbe auf schwarz setzen
  call graf (0.0, 5.0, 0.0, 0.5, -1.0, 1.0, -1.0, 0.1) !
 2D-Koordinatensystem setzen
  call curve (x, y, 100)                                  ! Graphen
 zeichnen

  ! *** Aufräumen ***
  call disfin                                             ! DISLIN
 beenden
end program dbsp2
```

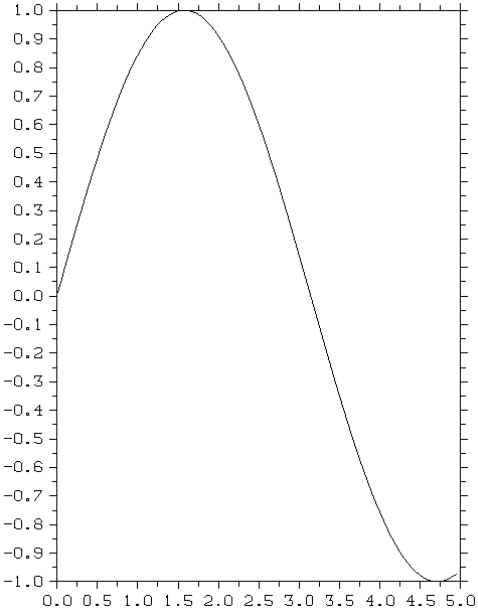


Abb. 49

43.3. Beispiel 3: Ein Pie-Chart

Fortran 90/95-Code (free source form)

```
program dbsp3
  implicit none

  real, dimension(3)    :: part = (/5.5, 2.5, 1.0/)
  integer, dimension(3) :: partcol1 = (/10, 100, 150/)
  integer, dimension(3) :: partcol2 = (/10, 100, 150/)

  integer :: setrgb

  ! *** Initialisierung ***
```

```
  call setpag ("DA4P")                                    ! DIN-A4
  call metafl ("CONS")                                    ! Console
 (Bildschirm)
  call disini                                             ! DISLIN
 initialisieren

  ! *** Zeichnen ***
  call pagfll (255)                                       !
 Hintergrundfarbe auf weiß setzen
  call color (setrgb (0., 0., 0.))                        !
 Vordergrundfarbe auf schwarz setzen
  call shdpat (16)                                        !
 Shadingpattern (16 = voll)
  call chnpie ("NONE")                                    ! Farbe und
 Shadingpattern
  call pieclr (partcol1, partcol2, 3)                     ! Teilfarben
  call pietyp ("3D")                                      ! 3D
  call piegrf ("Hallo", 0 , part, 3)                      ! Pie-Chart
 zeichnen

  ! *** Aufräumen ***
  call disfin                                             ! DISLIN
 beenden
end program dbsp3
```

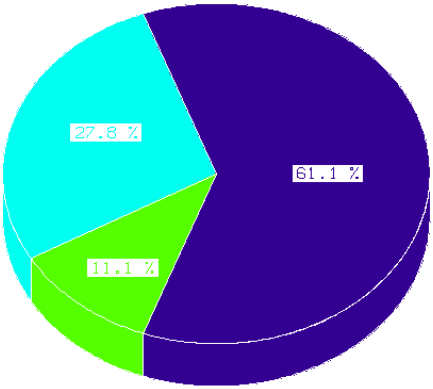


Abb. 50

43.4. Beispiel 4: Ein Meldungsfenster

Fortran 90/95-Code (free source form)

```
program dbsp4
  implicit none

  call disini
  call dwgmsg ("Hallo, Welt")
  call disfin
end program dbsp4
```

Abb. 51

Weitere (auch komplexere) Beispiele finden sich im ausführlichen *DISLIN*-Manual. Dieses ist auf der unten genannten Webpräsenz abrufbar.

44. Weblinks

- MAX-PLANCK-GESELLSCHAFT: DISLIN[1]

44.1. f03gl

44.2. Allgemeines

f03gl ist ein Fortran 2003-Interface für OPENGL[2], speziell für GLUT[3], freeglut und OpenGLUT. f03gl kann als der Nachfolger von F90GL[4] betrachtet werden.

44.3. Beispiel

Fortran 2003-Code

```
module bsp_ogl
  contains
    subroutine display()
      use opengl_gl
      use opengl_glut
```

---

1 HTTP://WWW.DISLIN.DE

2 HTTP://DE.WIKIPEDIA.ORG/WIKI/OPENGL

3 HTTP://DE.WIKIPEDIA.ORG/WIKI/OPENGL%20UTILITY%20TOOLKIT

4 HTTP://MATH.NIST.GOV/F90GL/

```
      implicit none
      call glclear(GL_COLOR_BUFFER_BIT + GL_DEPTH_BUFFER_BIT)
      call glColor3f( 0.2, 1.0, 0.3 )
      call glutSolidTeapot( 50.0_gldouble )
      call glutswapbuffers
    end subroutine display
    subroutine gfxinit
      use opengl_gl
      use opengl_glu
      implicit none
      real( glfloat ), dimension( 4 ) :: pos     = (/ 100.0, 100.0,
 200.0, 1.0 /)
      call glenable( GL_LIGHTING )
      call glenable( GL_LIGHT0 )
      call glenable( GL_DEPTH_TEST )
      call glenable( GL_COLOR_MATERIAL )
      call glenable( GL_NORMALIZE )
      call glenable( GL_POLYGON_SMOOTH )
      call gllightfv( GL_LIGHT0, GL_POSITION, pos )
      call glClearColor( 0.7, 0.7, 0.7, 0.0 )

      call glmatrixmode( GL_PROJECTION )
      call glOrtho( -100.0_gldouble, 100.0_gldouble, -100.0_gldouble,
 100.0_gldouble,  &
                   -100.0_gldouble, 100.0_gldouble );
      call glmatrixmode( GL_MODELVIEW )
      call glrotatef( 35.0, 1.0, 0.0, 0.0 )
      call glrotatef( -25.0, 0.0, 1.0, 0.0 )
    end subroutine gfxinit
end module bsp_ogl
program bsp
  use opengl_glut
  use bsp_ogl
  use, intrinsic :: iso_c_binding
  implicit none
  integer :: i
  call glutinit()
  call glutinitdisplaymode( GLUT_DOUBLE + GLUT_RGB + GLUT_DEPTH )
  i = glutcreatewindow( "Beispiel" // c_null_char )
  call gfxinit
  call glutdisplayfunc( display )
  call glutmainloop
end program bsp
```

Um das Beispiel nutzen zu können, müssen die Dateien *OpenGL_gl.f90*, *OpenGL_glu.f90* und wahlweise *OpenGL_glut,f90*,

OpenGL_freeglut.f90 oder *OpenGL_openglut.f90* von der Webseite „FORTRAN 2003 INTERFACE TO OPENGL"[5] herunter geladen werden. Die ersten beiden Dateien sind das Interface zu den beiden Bibliotheken GL (**G**raphics **L**ibrary) und GLU[6]. Die drei anderen Dateien sind das Interface für die Utility Toolkits GLUT[7], freeglut bzw. OpenGLUT.

Diese Dateien müssen dann kompiliert werden, beispielsweise für die Verwendung der freeglut:

```
g95 -c OpenGL_gl.f90
g95 -c OpenGL_glu.f90
g95 -c OpenGL_freeglut.f90
```

Kompilieren und Linken des Beipielprogrammses *bsp.f03*, hier als Beispiel, wenn sich die opengl_*.mod-Dateien und OpenGL_-*.o-Dateien im gleichen Verzeichnis wie die Beispieldatei *bsp.f03* befinden und die freeglut-Bibliothek verwendet wird (wie in einigen LINUX[8]-Distrubtionen wie z. B. RED HAT[9]):

```
g95 -o bsp bsp.f03 OpenGL_gl.o OpenGL_glu.o OpenGL_freeglut.o -lGL
-lGLU -lglut
```

Die Parameter *-lGL*, *-lGLU* geben die Pfade der beiden Bibliotheken GL und GLU an, während *-lglut* den Pfad des Utility Toolkits angibt.

5 HTTP://WWW-{}STONE.CH.CAM.AC.UK/PUB/F03GL/
6 HTTP://DE.WIKIPEDIA.ORG/WIKI/OPENGL%20UTILITY%20LIBRARY
7 HTTP://DE.WIKIPEDIA.ORG/WIKI/OPENGL%20UTILITY%20TOOLKIT
8 HTTP://DE.WIKIPEDIA.ORG/WIKI/LINUX
9 HTTP://DE.WIKIPEDIA.ORG/WIKI/RED%20HAT%20LINUX

Für Mac OS X ist folgende Sequenz zu verwenden:

```
g95 -o bsp bsp.f03 OpenGL_gl.o OpenGL_glu.o OpenGL_glut.o -framework
Carbon -framework OpenGL -framework GLUT
```

Nach der Eingabe von:

```
./bsp
```

sollte als Ausgabe die UTAH-TEEKANNE[10] erscheinen:

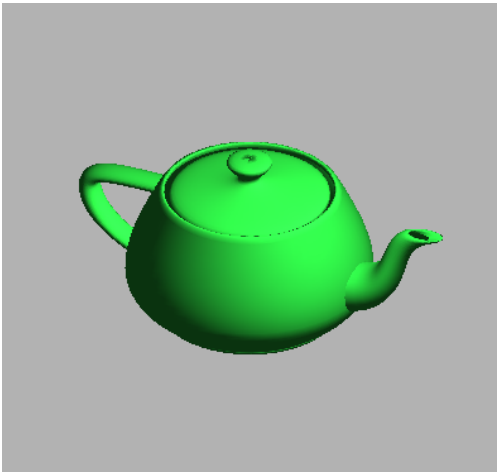


Abb. 52

10 HTTP://DE.WIKIPEDIA.ORG/WIKI/UTAH-{}TEEKANNE

44.4. Besonderheiten

Die Routinen für OpenGl und die GLUT sind in C geschrieben und auch die meisten Anwendungen von OpenGL und GLUT erfolgen in C. Daraus ergeben sich einige Besonderheiten für die Einbindung von OpenGL und GLUT unter Fortran.

In C lässt sich folgender Code erzeugen:

```
int main( int argc, char* argv[])
   {
   ...
   glutReshapeFunc(ChangeSize);
   ...
   return 0;
   }
```

```
void ChangeSize(GLsizei w, GLsizei h)
   {
   ...
   }
```

Dabei ist *GLsizei* ein von OpenGL spezifizierter Datentyp vom Typ *integer*.

In Fortran 2003 sieht das selbe Code-Fragement unter Verwendung der C-Interoperabilität von Fortran 2003 zum Beispiel folgenderweise aus:

Fortran 2003-Code

```
module simple_opengl_things
   use opengl_gl
   use opengl_glu
   use opengl_glut
   contains
      ...
      subroutine ChangeSize(w, h) bind(c)
         implicit none
         integer( kind=GLsizei ), value :: w
```

```
         integer( kind=GLsizei ), value :: h
         ...
      end subroutine ChangeSize
      ...
end module simple_open_gl_things
program simple_opengl
   use opengl_glut
   use simple_opengl_things
   ...
   call glutReshapeFunc( ChangeSize )
   ...
end program simple_opengl
```

44.5. Weblinks

- OpenGL[11]
- Fortran 2003 Interface to OpenGL[12]
- freeglut[13]
- OpenGLUT[14]
- Manuel für GLUT version 3[15]

44.6. Japi

11 http://www.opengl.org/
12 http://www-{}stone.ch.cam.ac.uk/pub/f03gl/
13 http://freeglut.sourceforge.net/
14 http://openglut.sourceforge.net/
15 http://www.opengl.org/documentation/specs/glut/spec3/spec3.html

45. Allgemeines

Auch für den Bereich des Graphical-User-Interface-Building sind Fortran-Bibliotheken verfügbar. Einerseits gibt es kommerziell-proprietäre Bibliotheken wie Interacter, Winteracter oder GinoMenu, die allerdings eindeutig auf rein professionellen Einsatz abzielen und deren Preise auch dementsprechend hoch liegen.

Hier soll darum anhand eines einfachen Beispiels auf eine mögliche Open-Source-Alternative zu diesen kommerziellen Bibliotheken hingewiesen werden, nämlich *japi*. *japi* steht unter der GNU Lesser General Public License und ist ein JAVA AWT[1]-Wrapper. *japi* ist für verschiedene Programmiersprachen erhältlich, so auch für FORTRAN 77. Mit kleineren Adaptierungen kann diese Bibliothek aber auch mit Fortran 90/95 verwendet werden. Mit *japi* ist derzeit nur ein Teil der AWT-Möglichkeiten abrufbar. Der Einsatz von *japi* setzt zwingend eine aktuelle Java-Installation (JRE oder JDK) auf dem Computer voraus.

1 HTTP://DE.WIKIBOOKS.ORG/WIKI/JAVA%20STANDARD%3A%20GRAFISCHE%20OBERFL%E4CHEN%20MIT%20AWT%20

46. japi-Installation

1. Download der Dateien "japi.f" und "libjapi.zip" von der japi-Homepage.
2. Entpacken der "libjapi.zip" (die Linux-Version enthält z.B. nur die "libjapi.a"-Bibliotheksdatei).
3. Verschieben der Bibliotheksdatei in ein geeignetes Verzeichnis, z.B. unter Linux in "/usr/lib" oder "/usr/local/lib".

47. Beispiel

Fortran 90/95-Code (free source form)

```
program jbsp
  include "japi.f95"

  integer               :: frame, obj
  integer, dimension(5) :: button

  if( .not. j_start()) then
    write(*,*) "JAPI-Problem"
    call end ()
  end if

  frame = j_frame("JAPI-Beispiel")

  call j_setborderlayout(frame)

  button(1) = j_button(frame, "Button 1")
  button(2) = j_button(frame, "Button 2")
  button(3) = j_button(frame, "Button 3")
  button(4) = j_button(frame, "Button 4")
  button(5) = j_button(frame, "Button 5")

  call j_setborderpos(button(1), J_LEFT)
  call j_setborderpos(button(2), J_RIGHT)
  call j_setborderpos(button(3), J_TOP)
  call j_setborderpos(button(4), J_BOTTOM)

  call j_show(frame)

  do
    obj=j_nextaction()

    if(obj == frame) call end ()
    if(obj == button(1)) write (*,*) "Button 1 gedrückt"
    if(obj == button(2)) write (*,*) "Button 2 gedrückt"
    if(obj == button(3)) write (*,*) "Button 3 gedrückt"
```

```
    if(obj == button(4)) write (*,*) "Button 4 gedrückt"
    if(obj == button(5)) write (*,*) "Button 5 gedrückt"
  end do

  call end ()
end program jbsp


subroutine end
  call j_quit()
  stop
end subroutine end
```

Die Include-Datei "japi.f95" ist eine adaptierte "japi.f"-Datei. "japi.f" ist als Teil der japi-Bibliothek im FORTRAN 77-Format auf der unten genannten *japi*-Webseite zu finden. Für eine Minimalanpassung müssen nur die C-Kommentarzeichen aus FORTRAN 77 gegen die !-Kommentarzeichen von Fortran 95 ausgewechselt werden.

Kompilieren, Linken:

```
gfortran -o jbsp jbsp.f95 -ljapi
```

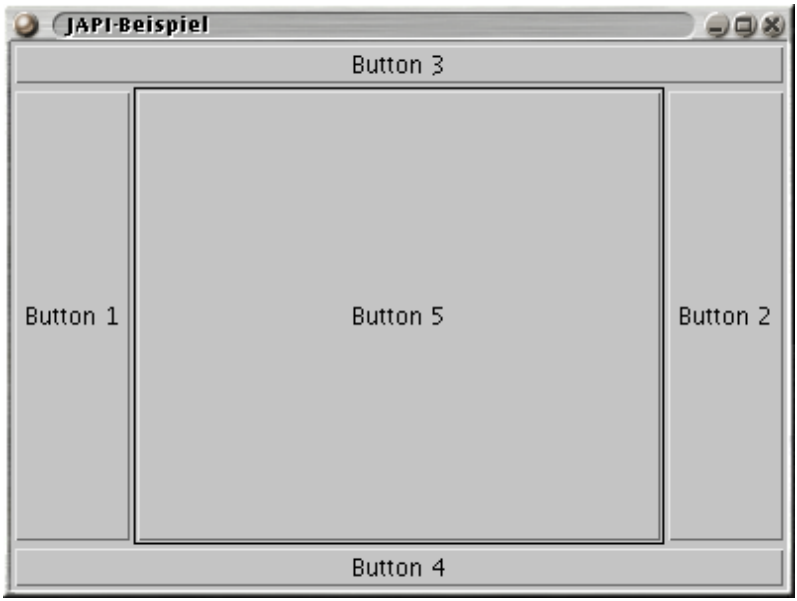


Abb. 53

Reference Manuals, sowie Programming Manuals zu *japi* sind auf der nachfolgend angeführten Homepage in verschiedenen Dateiformaten abrufbar.

48. Weblinks

- JAVA APPLICATION PROGRAMMING INTERFACE[1]

48.1. Pilib

1 HTTP://WWW.JAPI.DE

49. Allgemeines

Auch *pilib* ist ein Open-Source-Ansatz für die Erstellung von GUIs mittels Fortran. Anders als *japi* verwendet *pilib* zu diesem Zweck die GTK+-Bibliothek. Momentan befindet sich dieses Projekt in einer frühen Entwicklungsphase (Alpha-Status, Stand: Anfang 2006).

Für nähere Informationen hinsichtlich der GTK+-Bibliothek wird auf die GTK+-Homepage verwiesen.

50. pilib-Installation

1. Download des *pilib*-Softwarepakets von der im Abschnitt Weblinks angegebenen *pilib*-Internetadresse.
2. Entpacken (*gunzip*, *tar*).
3. Installation der Bibliotheksbestandteile für Linux mit dem üblichen *./configure, make, make install*.

Für eine detailliertere Installationsanleitung wird auf die im Softwarepaket enthaltene INSTALL- und README-Datei, sowie das *pilib*-Manual im HTML-Format verwiesen.

51. Beispiel

Fortran 90/95-Code (free source form)

```
module bspmod
  implicit none
  save

  integer :: myedit1, myedit2, myedit3
end module bspmod


program bsp
  use pimod
  use bspmod
  implicit none

  integer :: mywin, mycontainer, mybutton, mytext, myclose, myclick

  call piinit

  call gkwindow(c("Addition"), 1, 0, mywin, myclose)

  ! Container (in diesem Fall eine Table)
  call gkcontain(3, 2, 4, 5, mycontainer)
  call gkput(0, 0, -1, -1, mywin, mycontainer)

  ! Label
  call gktext(c("Zahl 1: "), mytext)
  call gkputtable(0, 0, 0, 0, 4, 4, 5, 5, -1, -1, mycontainer, mytext)


  ! Einzeiliges Eingabefeld mit einer Breite von 10 Zeichen
  call gkxedt(10, myedit1)
  call gkputtable(1, 0, 1, 0, 4, 4, 5, 5, -1, -1, mycontainer,
 myedit1)

  ! Label
  call gktext(c("+"), mytext)
```

```
  call gkputtable(0, 1, 1, 1, 4, 4, 5, 5, -1, -1, mycontainer, mytext)


  ! Label
  call gktext(c("Zahl 2: "), mytext)
  call gkputtable(0, 2, 0, 2, 4, 4, 5, 5, -1, -1, mycontainer, mytext)


  ! Einzeiliges Eingabefeld mit einer Breite von 10 Zeichen
  call gkxedt(10, myedit2)
  call gkputtable(1, 2, 1, 2, 4, 4, 5, 5, -1, -1, mycontainer,
 myedit2)

  ! Schaltfläche
  call gkbutton(c("="), mybutton, myclick)
  call gkputtable(0, 3, 2, 3, 4, 4, 5, 5, -1, -1, mycontainer,
 mybutton)

  ! Label
  call gktext(c("Ergebnis: "), mytext)
  call gkputtable(0, 4, 0, 4, 4, 4, 5, 5, -1, -1, mycontainer, mytext)


  !  Einzeiliges Eingabefeld mit einer Breite von 10 Zeichen
  call gkxedt(10, myedit3)
  call gkputtable(1, 4, 1, 4, 4, 4, 5, 5, -1, -1, mycontainer,
 myedit3)

  call gkshow(mywin)

  do while(myclose == 0)
    call gkproc

    if(myclick /= 0) then
      call calculate
      myclick = 0
    end if
  end do

  call gkdestroy(mywin)
end program bsp


subroutine calculate
  use pimod
  use bspmod
  implicit none
```

```
  real          :: k1, k2, string2real
  character(30) :: cstr

  k1 = string2real(myedit1)
  k2 = string2real(myedit2)

  write(cstr, *) k1+k2

  call gksetstring (c(cstr), myedit3)
end subroutine calculate


function string2real(widget)
  use pimod
  implicit none

  real                :: string2real, zahl
  integer, intent(in) :: widget
  character(30)       :: cstr
  type(string)        :: str

  call gkgetstring(str, widget)
  cstr = f_str2char(str)
  read(cstr, *) zahl   ! Umwandlung eines character-Wertes in eine
 real-Zahl
                       ! unter Zuhilfenahme des
 internal-file-Mechanismus
  string2real = zahl
end function string2real
```

Kompilieren, Linken:

```
g95 bsp.f95 -lpilib -lpilibf -I/usr/local/include
```

Bei der *pilib*-Installation werden *mod*-Dateien in ein Standard-Include-Verzeichnis geschrieben. Der Optionsschalter "-I" weist den Compiler an, im gegebenem Verzeichnis nach Include-Dateien zu suchen, in diesem Fall nach *mod*-Dateien. Das Format der mod-Dateien ist compilerabhängig.

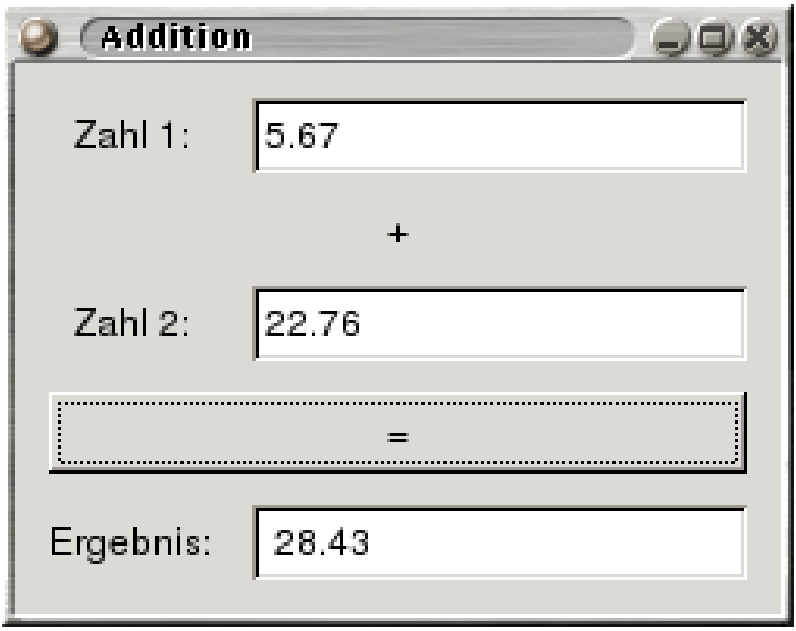


Abb. 54

Dieses Beispiel soll nur einen ersten Eindruck von *pilib* geben. Eine genauere Beschreibung der verwendeten *pilib*-Unterprogramme und Subroutinenparameter, sowie eine Auflistung weiterer Möglichkeiten der *pilib*-Bibliothek wird hier mit Hinweis auf die dem *pilib*-Softwarepaket beiliegenden Dokumentationsdateien nicht getätigt.

52. Weblinks

- PILIB-WIKI[1]
- PILIB-FORUM[2]
- PILIB BEI SOURCEFORGE[3]
- PILIB-MANUAL[4]
- GTK+[5]

1 HTTP://PILIB.BETA-{}CENTAURI.DE
2 HTTP://FORUM.PILIB.BETA-{}CENTAURI.DE
3 HTTP://SOURCEFORGE.NET/PROJECTS/PILIB
4 HTTP://PILIB.SOURCEFORGE.NET/PILIB.HTML
5 HTTP://WWW.GTK.ORG

53. Mathematik

53.1. BLAS und ATLAS

Basic Linear Algebra Subprograms[1]

1 http://de.wikipedia.org/wiki/%20Basic%20Linear%20Algebra%20Subprograms

54. Allgemeines

Die Basic Linear Algebra Subprograms (BLAS) stellen eine Sammlung von Unterprogrammen für die Vektor- und Matrizenrechnung dar.

- Level 1: Skalar-Vektor-, Vektor-Vektor-Operationen
- Level 2: Matrix-Vektor-Operationen
- Level 3: Matrix-Matrix-Operationen

Die Automatically Tuned Linear Algebra Software (ATLAS) ist ein um einige LAPACK-Funktionen erweitetertes BLAS-Paket und bietet die Möglichkeit, automatisiert eine rechneroptimierte Algebra-Bibliothek zu erzeugen.

55. Installation von BLAS

BLAS wird in Form von Fortran-Quellcodedateien in einem gepackten tar-Paket zur Verfügung gestellt. Ein Makefile zur Generierung einer Bibliotheksdatei wird nicht mitgeliefert. Eine derartige Bibliotheksdatei kann aber einfach selbst erstellt werden. Eine Anleitung findet sich z.B. auf der *gfortran*-Dokumentationsseite. Die notwendigen Schritte sind:

1. blas.tgz downloaden
2. Dieses Paket in ein leeres Verzeichnis entpacken
3. Bibliothek erstellen ("shared library" oder "static library"):
   a) In Form einer "shared library":
      *gfortran -shared -O2 *.f -o libblas.so -fPIC*
   b) In Form einer "static library":
      *gfortran -O2 -c *.f*
      *ar cr libblas.a *.o*
4. Die daraus resultierende Bibliotheksdatei in ein geeignetes Verzeichnis verschieben (z.B. */usr/lib/* oder */usr/local/lib/*)

56. Beispiele

56.1. Beispiel: Die Level 1-Funktionen *sdot* und *dnrm2*

Fortran 90/95-Code (free source form)

```
program bsp
  implicit none

  real, dimension(3) :: a = (/2.,1.,-1./), b = (/5., -2., 1.5/)
  real               :: c, sdot
  real(kind=8)       :: d, dnrm2

  ! *** Skalarprodukt ***
  ! sdot: s ... REAL, dot ... Skalarprodukt (inneres Produkt)
  ! 1. Argument  ... Dimension des Vektors
  ! 2. und 4. A. ... die Vektoren
  ! 3. und 5. A. ... Inkrement (hier 1)
  c = sdot(3, a, 1, b, 1)
  write(*,*) c
  ! Ausgabe: 6.500000

  ! *** Norm des Vektors ***
  ! dnrm2: d ... DOUBLE PRECISION, nrm2 ... (euklidische) Norm
  ! 1. Argument: Dimension des Vektors
  ! 2. A.:       Vektor
  ! 3. A.:       Inkrement (hier 1)
  d = dnrm2(3, dble(a), 1)
  write(*,*) d
  ! Ausgabe: 2.44948974278318
end program bsp
```

Kompilieren und Linken:

```
gfortran bsp.f95 -lblas
```

56.2. Beispiel: Die Level 2-Subroutine *sger*

sger steht für:

- s ... REAL
- ge ... general matrix
- r ... rank 1 operation

Mathematisch ist damit folgende Operation gemeint:

$A \leftarrow \alpha \mathbf{x}\mathbf{y}^T + A$

wobei A eine mxn-Matrix ist, x und y stellen Vektoren dar.

Das nachfolgende Beispiel führt konkret folgende Rechnung aus:

$$A = \begin{bmatrix} 2.0 \\ 1.0 \end{bmatrix} \begin{bmatrix} 1.0 & 0.0 \end{bmatrix} + \begin{bmatrix} 1.0 & -1.0 \\ 2.0 & 7.0 \end{bmatrix}$$

Fortran 90/95-Code (free source form)

```
program bsp
  implicit none

  real, dimension(2)   :: x = (/2., 1./), y = (/1., 0./)
  real, dimension(2,2) :: a = reshape ( (/1., 2., -1., 7./), (/2, 2/)
 )

  call sger (2, 2, 1., x, 1, y, 1, a, 2)
  write(*,*) a
  ! Ausgabe:  3.000000     3.000000    -1.000000     7.000000
end program bsp
```

57. Weblinks

- BLAS (Basic Linear Algebra Subprograms)[1]
- Automatically Tuned Linear Algebra Software (ATLAS)[2]

57.1. FGSL

GNU Scientific Library[3]

57.2. Allgemeines

Bei der "GNU Scientific Library" (GSL) handelt es sich um eine in C geschriebene Bibliothek. Diese bietet Funktionen für ein weites Spektrum der Mathematik. Beispielhaft seien folgende Bereiche genannt:

- Komplexe Zahlen
- Lineare Algebra
- Polynome
- Statistik
- Fast Fourier Transformation (FFT)

---

1 http://www.netlib.org/blas/

2 http://math-{}atlas.sourceforge.net/

3 http://de.wikipedia.org/wiki/%20Gnu%20Scientific%20Library

- Numerische Differentiation
- Numerische Integration
- Gewöhnliche Differentialgleichungen
- IEEE Floating-Point-Arithmetik

FGSL ist ein Fortran-Interface für diese GSL-Bibliothek. FGSL selbst deckt nicht die komplette Funktionspalette von GSL ab, inzwischen sind aber auch lineare Algebra und die FFT-Funktionen Bestandteil von FGSL, auch wenn zu diesem Zweck der Einsatz der optimierten Bibliotheken LAPACK und FFTW empfohlen wird. Für einige Teilbereiche werden nicht alle in C implementierten Datentypen unterstützt.

FGSL wurde unter Verwendung einiger *Fortran 2003*-Sprachmerkmale erstellt. Die Einbindung von FGSL in eigene Programme setzt aus diesem Grunde einen entsprechenden Fortran-Compiler voraus. Der g95-Compiler erfüllt z.B. diese Voraussetzungen.

Derzeit ist die FGSL-Version 0.9.3 vom 1. Mai 2010 aktuell.

57.3. Beispiele

57.3.1. Beispiel: Datentypen, Potenzierung und mathematische Konstanten

C	Fortran

• include <stdio.h>
• include <gsl/gsl_math.h>
 	
int main (void) { double a;
double d = 5.0;
a = gsl_pow_2 (d) * M_PI_4;
printf ("Kreisflaeche = %f\n",
a); return 0; }
	program bsp use fgsl implicit none
	real(kind = fgsl_double) :: a
	real(kind = fgsl_double) :: d =
	5.0_fgsl_double
	a = d ** 2 * m_pi_4
	write(*, *) "Kreisflaeche = ", a
	end program bsp

Kompilieren, Linken: `gcc bsp.c -I/usr/local/include -L/usr/local/lib -lgsl -lgslcblas -lm`	Kompilieren, Linken: `g95 bsp.f03 -I/usr/local/include/g95 -L/usr/local/lib -lgsl -lfgsl_g95 -lgslcblas`

Der `kind`-Wert `fgsl_double` entspricht dem `c_double` aus dem `iso_c_binding`-Modul. FGSL kennt die speziellen `pow`-Funktionen aus GSL nicht, da Fortran ohnehin über einen eigenen Potenzierungsoperator verfügt. Neben der `m_pi_4`-Konstante (= $\pi/4$) kennt FGSL noch eine ganze Reihe anderer mathematischer Konstanten, z.B.:

`m_e`	...	e, Eulersche Zahl, 2,714...
`m_euler`	...	Eulersche Konstante, 0,577...
`m_pi`	...	π
`m_pi_2`	...	$\pi/2$
`m_sqrt2`	...	$\sqrt{2}$

Auch jede Menge physikalische Konstanten kennt FGSL, z.B.:

`fgsl_const_mksa_speed_of_light`	...	Lichtgeschwindigkeit im Vakuum, 2.9979... x 10^8 m / s
`fgsl_const_mksa_molar_gas`	...	Allgemeine Gaskonstante, 8.314472 J / (K · mol)
`fgsl_const_mksa_inch`	...	0.0254m
`fgsl_const_mksa_torr`	...	133.322... Pa
`fgsl_const_num_giga`	...	10^9

57.3.2. Beispiel: Lösen einer quadratischen Gleichung

Gesucht ist die Lösung der quadratischen Gleichung $x^2 + 12x + 37 = 0$

C	Fortran
• include <stdio.h> • include <gsl/gsl_-complex.h> • include <gsl/gsl_poly.h> int main (void) { double a, b, c; gsl_complex z1, z2; int i; a = 1.0; b = 12.0; c = 37.0; i = gsl_poly_complex_solve_-quadratic(a, b, c, &z1, &z2); if(i == 1) { /* nur eine Lsg.*/ printf("z = (%f, %f)\n", z1.dat[0], z1.dat[1]); } else { /* 2 Lsg., reell oder komplex */ printf("z1 = (%f, %f)\n", z1.dat[0], z1.dat[1]); printf("z2 = (%f, %f)\n", z2.dat[0], z2.dat[1]); } return 0; /* Ausgabe: z1 = (-6.000000, -1.000000) z2 = (-6.000000, 1.000000) • / }	program bsp use fgsl implicit none real(kind = fgsl_double) :: a, b, c complex(fgsl_double), dimension(2) :: z integer(kind = fgsl_int) :: i a = 1.0_fgsl_double b = 12.0_-fgsl_double c = 37.0_fgsl_-double i = fgsl_poly_complex_solve_-quadratic(a, b, c, z(1), z(2)) if(i == 1) then ! nur eine Lsg. write(*, *) " z =", z(1) else ! 2 Lsg., reell oder komplex write(*, *) " z1,2 =", z end if ! Ausgabe: ! z1,2 = (-6.,-1.) (-6.,1.) end program bsp

57.3.3. Beispiel: Numerische Integration

Gesucht ist die Lösung des Integrals $I = \int_0^1 \frac{\sin x}{x} dx$

C	Fortran
• include <stdio.h> • include <math.h> • include <gsl/gsl_-integration.h> double f(double x, void *params) { return(sin(x) / x); } int main () { double result, error; size_t neval; gsl_function func; func.function = &f; func.params = 0; gsl_integration_qng (&func, 0.0, 1.0, 1e-9, 1e-9, &result, &error, &neval); printf ("Ergebnis = %f\n", result); return 0; /* Ausgabe: Ergebnis = 0.946083 • / }	module integral implicit none contains function f(x, params) bind(c) use, intrinsic :: iso_-c_binding implicit none real(kind = c_double) :: f real(kind = c_double), value :: x type(c_ptr), value :: params f = sin(x) / x end function f end module integral program bsp use fgsl use integral use, intrinsic :: iso_c_-binding implicit none real(kind = fgsl_double) :: result, error integer(kind = fgsl_size_t) :: neval integer(kind = fgsl_int) :: i type(fgsl_-function) :: func func = fgsl_function_init(f, c_null_ptr) i = fgsl_integration_qng (func, & 0.0_fgsl_double, & 1.0_fgsl_double, & 1e-9_fgsl_-double, & 1e-9_fgsl_double, & result, error, neval) write(*, *) "Ergebnis =", result ! Ausgabe: ! Ergebnis = 0.946083070367183 end program bsp

(F)GSL stellt verschiedene Möglichkeiten der numerischen Integration zur Verfügung. Hier wurde die Funktion für den QNG-Algorithmus (non-adaptive Gauss-Kronrod) gewählt.

57.3.4. Beispiel: IEEE-Floating-Point-Arithmetik

Darstellung einer Fließkommazahl nach IEEE 754-Standard:

$$(-1)^s(1.ffff...)2^E$$

Beispielsweise wird eine 32-bit-Fließkommazahl binär so aufgegliedert:

```
seeeeeeeeffffffffffffffffffffff
```

- s ... sign, 1 bit
- e, E ... exponent, 8 bit (2^8 = 256; E_{min} = -127; E_{max} = 128)
- f ... fraction, 23 bit

Näheres zum IEEE 754-Standard findet sich z.B. bei IEEE 754[4]

FGSL bietet Subroutinen, um Fießkommazahlen anschaulich entsprechend dem IEEE 754-Standard auszugeben:

- `fgsl_ieee_printf( x )` ... Ausgabe der Zahl x im IEEE-Format auf `stdout`
- `fgsl_ieee_fprintf( str, x )` ... Ausgabe der Zahl x im IEEE-Format. `str` ist ein C-Zeiger (C: `FILE *`, Fortran: `type( c_ptr )`).

4 HTTP://DE.WIKIPEDIA.ORG/WIKI/IEEE%20754

C	Fortran
• include <gsl/gsl_ieee_-utils.h> int main() { float a = 1.5; double b = -2.6666666666666666; gsl_ieee_printf_float(&a); gsl_ieee_printf_double(&b); }	program bsp use fgsl implicit none real(kind = fgsl_float) :: a = 1.5 real(kind = fgsl_double) :: b = -2.6666666666666666d0 call fgsl_ieee_printf(a) call fgsl_ieee_printf(b) end program bsp
Ausgabe: 1.10000000000000000000000*2^0 -1.01010101010101010101010101010101010101010101010101010101*2^1	

Mit der Subroutine `fgsl_ieee_env_setup()` lassen sich über die Environment-Variable `GSL_IEEE_MODE` einige nützliche Attribute (Rundungsmodus etc.) festlegen.

C	Fortran
• include <stdio.h> • include <gsl/gsl_ieee_-utils.h> int main() { float a = 1.5; int i; gsl_ieee_env_setup(); for(i = 0; i < 5; i++) { a /= 0.11; printf("%f\n", a); } }	program bsp use fgsl implicit none real :: a = 1.5 integer :: i call fgsl_ieee_env_setup() do i = 0, 4 a = a / 0.11 write(*, *) a end do end program bsp

C	Fortran
Programmaufruf: GSL_IEEE_-MODE="round-to-nearest" ./a.out Ausgabe: GSL_IEEE_-MODE="round-to-nearest,trap-common" 13.636364 123.966942 1126.972168 10245.201172 93138.195312	Programmaufruf: GSL_IEEE_-MODE="round-to-nearest" ./a.out Ausgabe: GSL_IEEE_-MODE="round-to-nearest,trap-common" 13.636364 123.96695 1126.9723 10245.203 93138.21
Programmaufruf: GSL_IEEE_-MODE="round-up" ./a.out Ausgabe: GSL_IEEE_-MODE="round-up,trap-common" 13.636364 123.966949 1126.972290 10245.203125 93138.210938	Programmaufruf: GSL_IEEE_-MODE="round-up" ./a.out Ausgabe: GSL_IEEE_-MODE="round-up,trap-common" 13.636364 123.96695 1126.9723 10245.203 93138.21
Programmaufruf: GSL_-IEEE_MODE="round-down" ./a.out Ausgabe: GSL_IEEE_-MODE="round-down,trap-common" 13.636363 123.966934 1126.972046 10245.200195 93138.179688	Programmaufruf: GSL_-IEEE_MODE="round-down" ./a.out Ausgabe: GSL_IEEE_-MODE="round-down,trap-common" 13.636363 123.966934 1126.972 10245.2 93138.18

57.4. Weblinks

- GSL - GNU SCIENTIFIC LIBRARY[5]

5 HTTP://WWW.GNU.ORG/SOFTWARE/GSL/

- FGSL: A FORTRAN INTERFACE TO THE GNU SCIENTIFIC LIBRARY[6]

57.5. LAPACK

LAPACK[7]

57.6. Allgemeines

LAPACK steht für "Linear Algebra Package". LAPACK ist eine Bibliothek zwecks Lösung von

- linearen Gleichungssystemen
- LLS-Aufgaben
- Eigenwertproblemen
- Singulärwertproblemen

LAPACK ist in FORTRAN 77 geschrieben. Die daraus resultierende Namensbeschränkung auf eine maximale Länge von 6 Zeichen führt zu sehr kryptischen Unterprogrammbezeichnungen, z.B.

	D	**G**	**E**	**T**	**R**	**F**
Zeichenposition:	D	M1	M2	O1	O2	O3

Erläuterung der Zeichenpositionen:

6 HTTP://WWW.LRZ-{}MUENCHEN.DE/SERVICES/SOFTWARE/MATHEMATIK/GSL/FORTRAN/INDEX.HTML

7 HTTP://DE.WIKIPEDIA.ORG/WIKI/%20LAPACK

D1	Datentyp:		
	S	...	real
	D	...	double precision
	C	...	complex
	Z	...	double complex
M1, M2	Matrixtyp, z.B.		
	GE	...	generell
	DI	...	diagonal
	OR	...	orthogonal (reelle Zahlen)
	SY	...	symmetrisch
O1, O2, (O3)	Operation, z.B.		
	TRF	...	faktorisiere

Eine detailliertere und umfassendere Beschreibung des Funktionsumfanges und der Anwendungsmöglichkeiten der LAPACK-Bibliothek bietet der LAPACK USERS' GUIDE[8]. Die einzelnen Subroutinen inklusive Unterprogrammparameter sind zudem auch in den LAPACK-Sourcecode-Dateien ausführlich dokumentiert.

57.7. Beispiel: Lösen eines einfachen Gleichungssystems

Gegeben ist folgendes Gleichungssystem:

8 HTTP://WWW.NETLIB.ORG/LAPACK/LUG/INDEX.HTML

$$2x + y = 5$$

$$3x + y = 6$$

bzw. in Matrixschreibweise:

$$\begin{bmatrix} 2 & 1 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \end{bmatrix} \Leftrightarrow A\mathbf{x} = B$$

Gesucht sind die Unbekannten x und y:

Fortran 90/95-Code (free source form)

```
program bsp
  implicit none

  integer :: info
  real, dimension(2,2) :: a = reshape ( (/2.,3.,1.,1./), (/2,2/) )
  real, dimension(2)   :: b = (/5., 6./), ipiv

!  SUBROUTINE SGESV(N, NRHS, A, LDA, IPIV, B, LDB, INFO)
!  s ... real, ge ... general matrix type, sv ... solver
   call sgesv(2, 1, a, 2, ipiv, b, 2, info)

   write(*,*) "Lösung (x, y): ", b

   if(info == 0) then
     write(*,*) "Ergebnis OK"
   else
     write(*,*) "Ergebnis NOK"
   end if

!  Ausgabe: Lösung (x, y):     1.000000       3.000000
!  Ausgabe: Ergebnis OK
end program bsp
```

Kompilieren, Linken:

```
gfortran bsp.f95 -llapack -lblas
```

57.8. Beispiel: Inverse Matrix

Gegeben ist eine 3x3-Matrix

$$A = \begin{pmatrix} 3 & -2 & 1 \\ -3 & 5 & 0 \\ 2 & -1 & 2 \end{pmatrix}$$

die invertiert werden soll. Die Zahlenwerte dieser Matrix A entsprechen einem Beispiel aus *Bartsch: Mathematische Formeln, 21. Auflage, VEB Fachbuchverlag Leipzig, 1986, Seite 109*, ebenfalls zum Thema "Inverse Matrix".

Verwendet werden hierzu die beiden LAPACK-Subroutinen:

- SGETRF
  - S ... Datentyp: real
  - GE ... Matrixtyp: general
  - TRF ... Operation: LU-Faktorisierung (Dreiecksform)
- SGETRI
  - S ... Datentyp: real
  - GE ... Matrixtyp: general
  - TRI ... Operation: Invertierung einer LU-faktorisierten Matrix

Fortran 90/95-Code (free source form)

```
program bsp
  implicit none
  real, dimension( 3, 3 ) :: A
  integer, dimension( 3 ) :: ipiv
  real, dimension( 3 )    :: work
  integer                 :: m = 3, n = 3, lda = 3, lwork = 3, info


  A =  reshape( (/  3.0, -3.0,  2.0, -2.0,  5.0, -1.0, 1.0,  0.0,  2.0
  /),  &
               shape( A ) )

! LU-Faktorisierung (Dreieckszerlegung) der Matrix A
  call sgetrf( m, n, A, lda, ipiv, info )
! Inverse der LU-faktorisierten Matrix A
```

```
  call sgetri( n, A, lda, ipiv, work, lwork, info )

  write( *, * ) "Inverse Matrix Ai =", A
  write( *, * ) "Testweise wie im Bartsch-Beispiel, Ai = 1/11 * (", A
 * 11, ")"

  if( info == 0 ) then
    write( *, * ) "OK"
  else
    write( *, * ) "Nicht OK"
  end if
! Ausgabe:
!   Inverse Matrix Ai = 0.909091 0.54545456 -0.63636374 0.2727273
!     0.36363637 -0.090909116 -0.45454553 -0.2727273 0.81818193
!   Testweise wie im Bartsch-Beispiel, Ai = 1/11 * ( 10.000001 6.
 -7.000001
!     3.0000005 4. -1.0000002 -5.000001 -3.0000005 9.000001 )
!   OK
end program bsp
```

57.9. Weblinks

- LAPACK - LINEAR ALGEBRA PACKAGE[9]
- LAPACK SEARCH ENGINE[10]
- LAPACK95 - FORTRAN95 INTERFACE TO LAPACK[11]
- ERSTELLEN DER LAPACK-BIBLIOTHEK MIT *gfortran*[12]

9 HTTP://WWW.NETLIB.ORG/LAPACK/
10 HTTP://WWW.CS.COLORADO.EDU/~{}JESSUP/LAPACK/
11 HTTP://WWW.NETLIB.ORG/LAPACK95/
12 HTTP://GCC.GNU.ORG/WIKI/GFORTRANBUILD

58. Parallele Programmierung

58.1. OpenMP

OpenMP[1]

58.2. Was ist OpenMP

OpenMP ist die Abkürzung für "*Open* specifications for *M*ulti **P**rocessing*" und ist eine API für Fortran und C/C++, die zum Zwecke der* PARALLELEN PROGRAMMIERUNG[2] *mittels Shared-Memory-Ansatz für Mehrprozessor-Systeme erschaffen wurde.*

Durch Anwendung von Compiler-Direktiven und spezieller Unterprogramme wird die Abarbeitung von bestimmten Programmkonstrukten auf mehrere Threads aufgeteilt.

Der "Master Thread" mit der Nummer 0 ist in einem OpenMP-Programm standardmäßig immer aktiv. Der Programmierer bestimmt im Programmcode, wann eine Gabelung (*fork*) in mehrere Threads gefordert wird und wann das Ganze wieder in einen einzelnen Thread vereint werden soll (*join*).

OpenMP wird schon von vielen Fortran-Compilern unterstützt.

---

1 HTTP://DE.WIKIPEDIA.ORG/WIKI/%20OPENMP

2 HTTP://DE.WIKIPEDIA.ORG/WIKI/PARALLELE%20PROGRAMMIERUNG

58.3. Ein einfaches Beispiel

Fortran 90/95-Code (free source form)

```
program bsp
  use omp_lib
  implicit none

  ! fork
  !$omp parallel num_threads(3)
    ! Das nur 1x ausgeben (beim Master Thread)
    if( omp_get_thread_num() == 0 ) then
      write( *, * ) 'Insgesamt gibt es ', omp_get_num_threads(),
 'Thread(s)'
    end if
    ! Das bei jedem Thread ausgeben
    write( *, * ) 'Thread ', omp_get_thread_num(), 'ist aktiv'
  ! join
  !$omp end parallel
! Ausgabe:
!    Insgesamt gibt es            3 Thread(s)
!    Thread            0 ist aktiv
!    Thread            1 ist aktiv
!    Thread            2 ist aktiv
end program bsp
```

Unter Umständen muss das Modul `omp_lib` eingebunden werden. Dieses Module enthält die `interface` für die OpenMP-Routinen. Eine mögliche Form des OpenMP-Modus ist am Ende dieses Abschnittes angegeben.

Kompilieren und Linken des Beispielprogramms:

gfortran:	gfortran -fopenmp -o bsp bsp.f90
Intel Fortran Compiler:	ifort -openmp -o bsp bsp.f90

Erläuterung:

- OpenMP-Direktiven werden als Kommentare gekapselt. Bei Verwendung der "free source form" lautet der erste Direktiven-

Abschnitt (en. *sentinel*, dt. *Wächter*) immer `!$omp`. Groß-/Kleinschreibung spielt keine Rolle. Es folgt die Anweisung, dass sich nun das Programm gabeln soll (`parallel`). Die Anzahl der gewünschten Threads wird hier explizit mittels der Option `num_threads()` festgelegt.

- Verwendete OpenMP-Funktionen:
  - `omp_get_thread_num()` ... Aktuelle Thread-Nummer
  - `omp_get_num_threads()` ... Anzahl der Threads
- Beendet wird der parallele Programmteil mit `!$omp end parallel`

58.4. Thread-Erzeugung: Die `parallel`-Direktive

Wie im vorigen Beispiel bereits angedeutet, wird ein "fork" (die Threaderzeugung) immer mit der Direktive

```
!$omp parallel [optionen]
```

eingeleitet und mit

```
!$omp end parallel
```

beendet. Es kann hier auch eine Reihe von optionalen Steueranweisungen angegeben werden (siehe vorheriges Beispiel und nachfolgende Beispiele).

58.5. Thread-Anzahl bestimmen

- Festlegung im Rahmen der OpenMP-Direktive `!$omp parallel` über die Option `num_threads( nr )`
- Mittels OpenMP-Subroutinenaufruf vor dem *fork*: `call omp_set_num_threads( nr )`
- Festlegung in der Kommandozeile vor Ausführung des Programmes, z.B.: `export OMP_NUM_THREADS=nr`
- Default (normalerweise 1 Thread pro CPU)
- Dynamische Anpassung zur Programmlaufzeit per Run-Time-Environment: `call omp_set_dynamic( .true. )`

58.6. Sichtbarkeit/Gültigkeit von Daten

Aufgrund des Shared-Memory-Ansatzes werden Daten standardmäßig zwischen den Threads geteilt. Dieses Verhalten kann aber auch optional geändert werden. Mögliche Varianten für die `parallel`-Direktive:

- `shared` ... Solche Daten sind explizit in allen Threads sichtbar und gültig. Eine Änderung solcher Daten in einem Thread wirkt sich auf alle anderen Threads aus.
- `private` ... Solche Daten sind nur im aktuellen Thread sichtbar und gültig, sie werden beim Eintritt in den parallelen Programmabschnitt nicht speziell initialisiert. Änderungen dieser Werte wirken sich nicht auf nachfolgende serielle Programmteile aus.
- `firstprivate` ... Ähnlich wie `private`. Der Unterschied zu `private` ist, dass solcherart markierte Daten mit dem letztgültigen Wert aus dem vorhergehenden seriellen Programmabschnitt initialisiert werden.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  use omp_lib
  implicit none

  integer :: a, b, c, tnr

  a = 123
  b = 123
  c = 123
  ! Seriell
  write( *, * ) 'Seriell:'
  write( *, * ) 'a = ', a
  write( *, * ) 'b = ', b
  write( *, * ) 'c = ', c
  write( *, * ) '--------------------------------'

  call omp_set_num_threads( 3 )
  !$omp parallel shared( a ) private( b ) firstprivate( c )
    write( *, * ) 'Parallel:'

    tnr = omp_get_thread_num()  ! Aktuelle Threadnummer

    if( tnr == 0 ) then
      a = a + 5
      b = b + 5
      c = c + 5
    end if

    write( *, * ) 'a = ', a
    write( *, * ) 'b = ', b
    write( *, * ) 'c = ', c
    write( *, * ) '--------------------------------'
  !$omp end parallel
  ! Seriell
  write( *, * ) 'Seriell:'
  write( *, * ) 'a = ', a
  write( *, * ) 'b = ', b
  write( *, * ) 'c = ', c
end program bsp
```

Ausgabe:

```
 Seriell:
 a =          123
```

```
b =          123
c =          123
-------------------------------
Parallel:           0
a =          128
b =            5
c =          128
-------------------------------
Parallel:           1
a =          128
b =            0
c =          123
-------------------------------
Parallel:           2
a =          128
b =            0
c =          123
-------------------------------
Seriell:
a =          128
b =          123
c =          123
```

Für andere OpenMP-Direktiven sind auch noch andere Sichtbarkeits- und Gültigkeitsbereiche möglich (z.B. `lastprivate`).

58.7. Die `do`-Direktive

Innerhalb eines `parallel`-Blocks können auch `do`-Schleifen parallelisiert werden. Die Schleifendurchläufe werden auf die einzelnen Threads bzw. CPUs aufgeteilt.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  use omp_lib
  implicit none

  integer :: i, tnr
```

```
  call omp_set_num_threads( 3 )
  !$omp parallel private( i )
    !$omp do
      do i = 1, 20
        tnr = omp_get_thread_num()  ! Aktuelle Threadnummer
    write( *, * ) 'Thread', tnr, ':',  i
      end do
    !$omp end do
  !$omp end parallel
end program bsp
```

Ausgabe:

```
 Thread           0 :           1
 Thread           0 :           2
 Thread           0 :           3
 Thread           0 :           4
 Thread           0 :           5
 Thread           0 :           6
 Thread           0 :           7
 Thread           1 :           8
 Thread           1 :           9
 Thread           1 :          10
 Thread           1 :          11
 Thread           1 :          12
 Thread           1 :          13
 Thread           1 :          14
 Thread           2 :          15
 Thread           2 :          16
 Thread           2 :          17
 Thread           2 :          18
 Thread           2 :          19
 Thread           2 :          20
```

Die Zuweisung der Schleifendurchläufe an die Threads kann gesteuert werden. Dazu wird der `do`-Direktive eine `schedule`-Anweisung mit dem Argument Typ und ev. auch mit dem Argument Chunk-Größe beigefügt. Als Typen sind möglich

- `static`
- `dynamic`

- `guided`
- `runtime`

Diese Bezeichnungen beziehen sich auf die Art der Thread-Erzeugung.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  use omp_lib
  implicit none

  integer :: i, tnr

  call omp_set_num_threads( 3 )
  !$omp parallel private( i )
    !$omp do schedule(static, 3)
      do i = 1, 20
        tnr = omp_get_thread_num()  ! Aktuelle Threadnummer
    write( *, * ) 'Thread', tnr, ':',  i
      end do
    !$omp end do
  !$omp end parallel
end program bsp
```

Ausgabe:

```
 Thread           0 :           1
 Thread           0 :           2
 Thread           0 :           3
 Thread           0 :          10
 Thread           0 :          11
 Thread           0 :          12
 Thread           0 :          19
 Thread           0 :          20
 Thread           1 :           4
 Thread           1 :           5
 Thread           1 :           6
 Thread           1 :          13
 Thread           1 :          14
 Thread           1 :          15
 Thread           2 :           7
 Thread           2 :           8
```

```
Thread           2 :           9
Thread           2 :          16
Thread           2 :          17
Thread           2 :          18
```

Eine do while-Schleife kann nicht auf diese Art und Weise mittels OpenMP-Direktive parallel ausgeführt werden.

58.8. Die sections-Direktive

Auch die Festlegung, dass bestimmte Programmabschnitte auf je einen Thread verteilt werden sollen, ist möglich. Dazu wird das Konstrukt

```
!$omp sections [optionen]
  !$omp section
    block
  !$omp section
    block
  ...
!$omp end sections
```

innerhalb eines parallel-Blocks eingesetzt.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  use omp_lib
  implicit none

  integer :: a, b

  a = 20
  b = 30

  call omp_set_num_threads( 3 )
  !$omp parallel shared( a, b )
    !$omp sections
```

```
      !$omp section
        write( *, * ) omp_get_thread_num(), a
        write( *, * ) omp_get_thread_num(), "---"
      !$omp section
        write( *, * ) omp_get_thread_num(), b
        write( *, * ) omp_get_thread_num(), "----"
    !$omp end sections
  !$omp end parallel
! Ausgabe (ifort):
!   0          20
!   0 ---
!   1          30
!   1 ----
end program bsp
```

58.9. Weitere Direktiven

- `workshare`
- `single`

58.10. Kombinierte Direktiven

Unmittelbar aufeinanderfolgende Einzeldirektiven können auch in einer einzigen Direktive zusammengefasst werden. Möglich sind

- `parallel do`
- `parallel sections`
- `parallel workshare`

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  use omp_lib
  implicit none

  integer :: i, tnr
```

```
  call omp_set_num_threads( 3 )
  !$omp parallel do private(i) schedule(static, 3)
      do i = 1, 20
        tnr = omp_get_thread_num()  ! Aktuelle Threadnummer
    write( *, * ) 'Thread', tnr, ':',  i
      end do
  !$omp end parallel do
end program bsp
```

Eine zusammengehörende OpenMP-Direktive darf auch auf mehrere Zeilen verteilt werden.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  use omp_lib
  implicit none

  integer :: i, tnr

  call omp_set_num_threads( 3 )
  !$omp parallel do &
  !$omp private(i) &
  !$omp schedule(static, 3)
    do i = 1, 20
      tnr = omp_get_thread_num()  ! Aktuelle Threadnummer
      write( *, * ) 'Thread', tnr, ':',  i
    end do
  !$omp end parallel do
end program bsp
```

58.11. Synchronisation

Bei der parallelen Programmierung können Situationen auftreten, die bei einer seriellen Programmausführung nie passieren würden, z.B. RACE CONDITIONS[3]. Damit es nicht soweit kommt, bietet

3 HTTP://DE.WIKIPEDIA.ORG/WIKI/RACE%20CONDITION

OpenMP einige Direktiven zur Synchronisation der Threadausführung.

58.11.1. `master`-Direktive

```
!$omp master
  ...
!$omp master end
```

Der eingeschlossene Programmblock wird nur vom Master Thread ausgeführt und von den anderen Threads ignoriert.

58.11.2. `critical`-Direktive

```
!$omp critical
  ...
!$omp critical end
```

Dieser Programmteil wird zwar von allen Threads ausgeführt, allerdings ist sichergestellt, dass dies nicht gleichzeitig erfolgt.

58.11.3. `atomic`-Direktive

```
!$omp atomic
```

Ähnlich zu `critical`. Allerding gilt dies Direktive nur für eine einzelne unmittelbar nachfolgende spezielle Programmanweisung.

58.11.4. `barrier`-Direktive

```
!$omp barrier
```

Sobald ein Thread eine solche Barriere erreicht, wartet er bis alle andere Threads diese Barriere auch erreicht haben. Erst dann geht's weiter.

58.11.5. `flush`-Direktive

```
!$omp flush
```

Erstellung eines konsistenten Speicherbildes.

58.11.6. `ordered`-Direktive

```
!$omp do ordered
  do ...
    ...
    !$omp ordered
      ...
    !$omp end ordered
    ...
  end do
!$omp end do
```

"Geordnete Ausführung" von `do`-Schleifen in der gleichen Reihenfolge einer seriellen Abarbeitung.

Beispiel: **Fortran 90/95-Code (free source form)**

```
program bsp
  use omp_lib
  implicit none

  integer :: i, tnr
```

```
  call omp_set_num_threads( 3 )
  !$omp parallel private( i )
    !$omp do ordered schedule(static, 3)
      do i = 1, 20
        !$omp ordered
      tnr = omp_get_thread_num()  ! Aktuelle Threadnummer
          write( *, * ) 'Thread', tnr, ':',  i
        !$omp end ordered
      end do
    !$omp end do
  !$omp end parallel
end program bsp
```

Ausgabe:

```
 Thread           0 :           1
 Thread           0 :           2
 Thread           0 :           3
 Thread           1 :           4
 Thread           1 :           5
 Thread           1 :           6
 Thread           2 :           7
 Thread           2 :           8
 Thread           2 :           9
 Thread           0 :          10
 Thread           0 :          11
 Thread           0 :          12
 Thread           1 :          13
 Thread           1 :          14
 Thread           1 :          15
 Thread           2 :          16
 Thread           2 :          17
 Thread           2 :          18
 Thread           0 :          19
 Thread           0 :          20
```

58.12. Das Modul omp_lib

Das Modul omp_lib enthält die interface für die Routinen von OpenMP. Eine mögliche Form des Modules ist nachfolgend abgebildet. In dem Modul wird die import-Anweisung verwendet, die Teil des Standards *Fortran 2003* ist.

Fortran 90/95-Code (free source form)

```
module omp_lib
!
! OpenMP Fortran API v2.5
!
    implicit none

    integer, parameter, private :: sgl = kind( 0.0 )
    integer, parameter, private :: dbl = kind( 0.0d0 )

    integer, parameter, private :: omp_real_kind = dbl
    integer, parameter, private :: omp_integer_kind = sgl
    integer, parameter, private :: omp_logical_kind = sgl
    integer, parameter, private :: omp_lock_kind = dbl
    integer, parameter, private :: omp_nest_lock_kind = dbl

    interface
        subroutine omp_destroy_lock ( var )
            import :: omp_lock_kind
            integer ( kind=omp_lock_kind ), intent(inout) :: var
        end subroutine omp_destroy_lock
        subroutine omp_destroy_nest_lock ( var )
            import :: omp_nest_lock_kind
            integer ( kind=omp_nest_lock_kind ), intent(inout) :: var
        end subroutine omp_destroy_nest_lock
        function omp_get_dynamic ()
            import :: omp_logical_kind
            logical ( kind=omp_logical_kind ) :: omp_get_dynamic
        end function omp_get_dynamic
        function omp_get_max_threads ()
            import :: omp_integer_kind
            integer ( kind=omp_integer_kind ) :: omp_get_max_threads
        end function omp_get_max_threads
        function omp_get_nested ()
            import :: omp_logical_kind
            logical ( kind=omp_logical_kind ) :: omp_get_nested
        end function omp_get_nested
```

```
        function omp_get_num_procs ()
            import :: omp_integer_kind
            integer ( kind=omp_integer_kind ) :: omp_get_num_procs
        end function omp_get_num_procs
        function omp_get_num_threads ()
            import :: omp_integer_kind
            integer ( kind=omp_integer_kind ) :: omp_get_num_threads
        end function omp_get_num_threads
        function omp_get_thread_num ()
            import :: omp_integer_kind
            integer ( kind=omp_integer_kind ) :: omp_get_thread_num
        end function omp_get_thread_num
        function omp_get_wtick ()
            import :: omp_real_kind
            real ( kind=omp_real_kind ) :: omp_get_wtick
        end function omp_get_wtick
        function omp_get_wtime ()
            import :: omp_real_kind
            real ( kind=omp_real_kind ) :: omp_get_wtime
        end function omp_get_wtime
        subroutine omp_init_lock ( var )
            import :: omp_lock_kind
            integer ( kind=omp_lock_kind ), intent(out) :: var
        end subroutine omp_init_lock
        subroutine omp_init_nest_lock ( var )
            import :: omp_nest_lock_kind
            integer ( kind=omp_nest_lock_kind ), intent(out) :: var
        end subroutine omp_init_nest_lock
        function omp_in_parallel ()
            import :: omp_logical_kind
            logical ( kind=omp_logical_kind ) :: omp_in_parallel
        end function omp_in_parallel
        subroutine omp_set_dynamic ( enable_expr )
            import :: omp_logical_kind
            logical ( kind=omp_logical_kind ), intent(in) ::
enable_expr
        end subroutine omp_set_dynamic
        subroutine omp_set_lock ( var )
            import :: omp_lock_kind
            integer ( kind=omp_lock_kind ), intent(inout) :: var
        end subroutine omp_set_lock
        subroutine omp_set_nest_lock ( var )
            import :: omp_nest_lock_kind
            integer ( kind=omp_nest_lock_kind ), intent(inout) :: var
        end subroutine omp_set_nest_lock
        subroutine omp_set_nested ( enable_expr )
            import :: omp_logical_kind
            logical ( kind=omp_logical_kind ), intent(in) ::
```

```
 enable_expr
        end subroutine omp_set_nested
        subroutine omp_set_num_threads ( number_of_threads_expr )
            import :: omp_integer_kind
            integer ( kind=omp_integer_kind ), intent(in) ::
 number_of_threads_expr
        end subroutine omp_set_num_threads
        function omp_test_lock ( var )
            import :: omp_logical_kind, omp_lock_kind
            logical ( kind=omp_logical_kind ) :: omp_test_lock
            integer ( kind=omp_lock_kind ), intent(inout) :: var
        end function omp_test_lock
        function omp_test_nest_lock ( var )
            import :: omp_integer_kind, omp_nest_lock_kind
            integer ( kind=omp_integer_kind ) :: omp_test_nest_lock
            integer ( kind=omp_nest_lock_kind ), intent(inout) :: var
        end function omp_test_nest_lock
        subroutine omp_unset_lock ( var )
            import :: omp_lock_kind
            integer ( kind=omp_lock_kind ), intent(inout) :: var
        end subroutine omp_unset_lock
        subroutine omp_unset_nest_lock ( var )
            import :: omp_nest_lock_kind
            integer ( kind=omp_nest_lock_kind ), intent(inout) :: var
        end subroutine omp_unset_nest_lock
    end interface

end module omp_lib
```

58.13. Literatur

- Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, Ramesh Menon, *Parallel Programming in OpenMP*, Morgan Kaufmann Publishers, 2001, ISBN-13: 978-1-55860-671-5, ISBN-10: 1-55860-671-8

58.14. Weblinks

- OpenMP[4]
- OpenMP-Homepage[5]
- OpenMP Fortran Summary[6]
- OpenMP Tutorial[7]
- OpenMp in gfortran[8]

4 http://de.wikipedia.org/wiki/OpenMP
5 http://www.openmp.org/drupal/
6 http://studies.ac.upc.edu/FIB/MP/openmp_f.pdf
7 http://www.llnl.gov/computing/tutorials/openMP/
8 http://karminghenry.sinaman.com/gfortran.html

Teil VII.

Fortran in Kombination mit anderen Programmiersprachen

59. Fortran und Tcl

Tcl/Tk kann im Zusammenhang mit Fortran zwecks Erstellung einer Tk-Benutzeroberfläche für Fortran-Programme interessant sein. Die zeitkritischen oder mathematisch orientierten Programmteile werden mittels Fortran-Code realisiert. Der Programmcode für die Benutzerschnittstelle wird mittels Tcl/Tk-Skript zur Verfügung gestellt.

59.1. Beispiel

59.1.1. Prinzipskizze

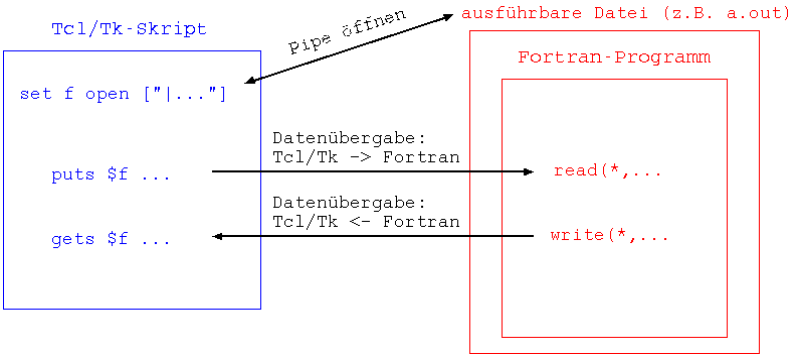


Abb. 55

59.1.2. Tcl/Tk-Code

```
#!/usr/bin/wish

wm title . Sinus                                    ;# Fenstertitel

entry  .e1                                          ;# Eingabefeld
button .b1 -text "Hier drücken" -command fcall      ;# Schaltfläche
label  .l1 -bg green                                ;# Textfeld

pack .e1 -padx 10 -pady 5                           ;# Widgets packen
pack .b1 -padx 10 -pady 5
pack .l1 -padx 10 -pady 5

proc fcall { } {                                    ;# Kommunikation mit
 Fortran, Ergebnis schreiben
  set f [open "|./a.out" r+]                        ;# a.out ist das
 kompilierte und gelinkte Fortran-Programm
  set val [.e1 get]

  puts $f $val
  flush $f

  gets $f wert
  close $f

  .l1 config -text $wert
}
```

59.1.3. Fortran-Code

Fortran 90/95-Code (free source form)

```
program bsp
  implicit none

  real :: val, sin

  read (*,*) val
  write(*,'(A12F6.3)') "Ergebnis = ", sin(val)
end
```

59.1.4. Programmausführung

Das Fortran-Programm muss selbstverständlich vorab einmal kompiliert und gelinkt werden. Im Beispielsfall muss die exekutierbare Ausgabedatei *a.out* heißen. Unter Linux wird das Tcl-Skript vor dem ersten Start als ausführbar (-> mittels *chmod*-Befehl) markiert. Der Programmaufruf erfolgt über das Tcl-Skript, das wie ein normales Programm durch Eingabe des Programmnamens gestartet wird.

59.1.5. Ergebnis

Abb. 56

59.2. Alternativen

Tcl/Tk bietet eine Schnittstelle zur Programmiersprache C (`tcl.h`, `tk.h`, `libtcl*.so`, `libtk*.so`). Mit dem im nächsten Kapitel behandelten Fortran-C-Binding kann auf diese C-Funktionen zugegriffen werden. Die Ftcl-Bibliothek nutzt diesen Mechanismus.

60. Weblinks

- Tcl/Tk-Homepage[1]
- Programming:Tcl (englischsprachiges Wikibook)[2]
- Tcl/Tk Cookbook - Tcl/Tk and FORTRAN[3]
- Ftcl: Using Tcl in Fortran programs and vice versa[4]

1 http://www.tcl.tk/
2 http://en.wikibooks.org/wiki/Programming:Tcl
3 http://carpanta.dc.fi.udc.es/docs/tcltk-{}cookbook/chap7.html
4 http://ftcl.sourceforge.net/

61. Fortran und C

61.1. Interface

Ein Interface ist die Schnittstellendefinition zu einem externen Unterprogramm. Das externe Unterprogramm kann, muss aber nicht in Fortran geschrieben sein.

interface Spezifikation der externen Unterprogramme end interface

Beispiel:

```
interface
  ! subroutine x(a, b, c) sei ein externes Unterprogramm
  subroutine x(a, b, c)
    real, intent(in) :: a
    integer          :: b, c
  end subroutine x
end interface

call x(2.5, 1, 3)
```

61.2. Fortran 90/95

In Fortran 90/95 ist der Zugriff auf C-Funktionen nicht standardisiert. Derartige Zugriffe sind compilerabhängig zu lösen. Es

sind je nach verwendetem Fortran-Compiler verschiedene Compilerdirektiven zu setzen, Optionen beim Compileraufruf und ähnliches zu beachten. Fortran 2003 bietet eine standardisierte Schnittstelle für den Zugriff auf C-Funktionen.

61.3. g95 (gfortran) und gcc

61.3.1. Beispiel: "call by value" und "call by reference"

Fortran übergibt die Argumente eines Unterprogrammes standardmäßig "call by reference". Im Zusammenspiel mit C besteht nun das Problem, dass in C Argumente "call by reference" via Pointer oder "call by value" übergeben werden . In *g77*, *g95* und *gfortran* ist die Funktion *%val* inkludiert, mit der auch in Fortran der "call by value"-Mechanismus nachgebildet werden kann.

Fortran-Code *bsp.f90*: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  interface
    subroutine zahl(a, b)
      integer :: a, b
    end subroutine zahl
  end interface

  call zahl(%val(5), 7)
end program bsp
```

C-Code *bsp.c*: **Programmcode**

```
#include <stdio.h>

void zahl(int a, int *b)
{
  printf("%s%d\n", "Ergebnis = ", a * *b);
}
```

Compilieren und Linken:

```
gcc -c -o bsp1.o bsp.c
g95 -c -fno-underscoring -o bsp2.o bsp.f90
g95 bsp1.o bsp2.o
```

Ausgabe:

```
Ergebnis = 35
```

61.3.2. Beispiel: Übergabe von Zeichenketten

In C sind Strings im Gegensatz zu Fortran Null-terminiert. Dies muss beim Aufruf einer C-Prozedur aus Fortran berücksichtigt werden.

Fortran Code *bsp.f90*: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  interface
    subroutine hallo(str)
      character(*) :: str
    end subroutine
  end interface

  call hallo("Hallo, Sonne" // char(0))        ! char(0) ->
 Nulltermination für C
  call hallo("Hallo, Protuberanz" // char(0))  ! char(0) ->
 Nulltermination für C
  call hallo("Hallo, Mond")                    ! keine Nulltermination
! Ausgabe:
!  Hallo, Sonne
!  Hallo, Protuberanz
!  Hallo, Mond ... More segments remain
end program bsp
```

C-Code *bsp.c*: **Programmcode**

```
#include <stdio.h>

void hallo(char *str)
{
  printf("%s\n", str);
}
```

Compilieren und Linken:

```
gcc -c -o bsp1.o bsp.c
g95 -c -fno-underscoring -o bsp2.o bsp.f90
g95 bsp1.o bsp2.o
```

61.3.3. Beispiel: Rückgabewert

Die Rückgabe eines Wertes einfachen Datentyps aus einer C-Funktion nach Fortran stellt kein Problem dar. Es ist einzig zu beachten, dass der Datentyp in C und Fortran übereinstimmt.

Fortran-Code *bsp.f90*: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  interface
    function zahl(x, y)
      integer :: zahl
      integer :: x, y
    end function
  end interface

  integer :: res

  res = zahl(%val(76), %val(32))
  write(*,*) res
!  Ausgabe: 108
end program bsp
```

C-Code *bsp.c*: **Programmcode**

```
int zahl(int a, int b)
{
  return (a+b);
}
```

61.4. ifort und gcc

61.4.1. Beispiel: "call by value" und "call by reference"

Fortran-Code *bsp.f90*: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  interface
    subroutine zahl(a, b)
      !dec$ attributes c :: zahl
      !dec$ attributes reference :: b
      integer :: a, b
    end subroutine zahl
  end interface

  call zahl(5, 7)
end program bsp
```

C-Code *bsp.c*: **Programmcode**

```
#include <stdio.h>

void zahl(int a, int *b)
{
  printf("%s%d\n", "Ergebnis = ", a * *b);
}
```

Compilieren, Linken:

```
gcc -c -o bsp1.o bsp.c
ifort -c -o bsp2.o bsp.f90
ifort bsp1.o bsp2.o
```

Ausgabe:

```
Ergebnis = 35
```

61.4.2. Beispiel: Übergabe von Zeichenketten

Fortran-Code *bsp.f90*: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  interface
    subroutine hallo(str)
      !dec$ attributes c :: hallo
      !dec$ attributes reference :: str
      character(*) :: str
    end subroutine
  end interface

  call hallo("Hallo, Sonne" // char(0))  ! char(0) -> Nulltermination
  call hallo("Hallo, Protuberanz"C)      ! C       -> Nulltermination
 (bei Intel-Fortran-Compiler)
  call hallo("Hallo, Mond")              ! keine explizite
 Nullterminiation
! Ausgabe:
!  Hallo, Sonne
!  Hallo, Protuberanz
!  Hallo, Mond
end program bsp
```

C-Code *bsp.c*: **Programmcode**

```
#include <stdio.h>

void hallo(char *str)
{
```

```
  printf("%s\n", str);
}
```

Compilieren und Linken:

```
gcc -c -o bsp1.o bsp.c
ifort -c -o bsp2.o bsp.f90
ifort bsp1.o bsp2.o
```

61.4.3. Beispiel: Rückgabewert

Fortran-Code *bsp.f90*: **Fortran 90/95-Code (free source form)**

```
program bsp
  implicit none

  interface
    function zahl(x, y)
      !dec$ attributes c :: zahl
      integer :: zahl
      integer :: x, y
    end function
  end interface

  integer :: res

  res = zahl(76, 32)
  write(*,*) res
!  Ausgabe: 108
end program bsp
```

C-Code *bsp.c*: **Fortran 90/95-Code (free source form)**

```
int zahl(int a, int b)
{
  return (a+b);
}
```

61.5. Fortran 2003

In *Fortran 2003* ist es viel einfacher auf *C* zu zugreifen, als in *Fortran 95*. Es wurde im *Fortran 2003*-Standard ein intrinsisches Modul namens `iso_c_binding` vorgesehen, das die zum Zugriff auf C-Programme nötigen Elemente enthält.

61.6. Ein einfaches Beispiel

Beispiel funktioniert mit Compiler

- g95 (0.91!) May 10 2007: ja
- gfortran 4.3.0 20070723 (experimental): ja
- Intel Fortran Compiler 10.0: ja
- Sun Studio Express - June 2007: ja

Anmerkungen:

Fortran 2003-Code: bsp.f95 **Fortran 2003-Code**

```
program bsp
  implicit none
  interface
    function addition( a, b ) bind( c[, name="c_func"] )
      use, intrinsic :: iso_c_binding
      real( kind = c_float ), value :: a
      real( kind = c_float ), value :: b
      real( kind = c_float )        :: addition
    end function addition
  end interface
  write (*,*) addition( 2.5, 3.3 )
! Ausgabe:  5.8
end program bsp
```

C-Code: bsp.c **Programmcode**

```
float addition(float a, float b)
{
```

```
  return (a + b);
}
```

Makefile: **Programmcode**

```
FC = g95  # oder gfortran, ...
CC = gcc  # oder icc, ...
bsp: bsp_c.o bsp_f95.o
    $(FC) -o bsp bsp_c.o bsp_f95.o
bsp_c.o: bsp.c
    $(CC) -c -o bsp_c.o bsp.c
bsp_f95.o: bsp.f95
    $(FC) -c -o bsp_f95.o bsp.f95
.PHONY: clean
clean:
    rm *.o
```

Was ist neu gegenüber Fortran 95?

- `bind( c[, name="c_func"] )` ... bind-Attribut, stellt unter anderem die Interoperabilität mit C bezüglich Prozedurnamenskonventionen nach der Übersetzung sicher. Mittels des optionalen Arguments "name" kann die Funktion in Fortran umbenannt werden. "c_func" ist dabei der Name der Funktion in C.
- `use, intrinsic :: iso_c_binding` ... Einbindung des intrinsischen Moduls `iso_c_binding`

... C-Datentyp `float`.

- `real( kind = c_float )`

  ... call by value

- `value`

61.7. Datentyp-Zuordnung

Das `iso_c_binding`-Modul stellt benannte Konstanten zur Verfügung, die bei Fortran-Datentypen als kind-Wert zu verwenden sind, um den jeweiligen C-Datentyp zu charakterisieren. Weist eine solche Konstante einen negativen Wert auf, dann ist keine Entsprechung von Fortran-Datentyp zu C-Datentyp vorhanden.

Fortran-Datentyp	**Benannte** `iso_c_binding`-**Konstante (kind-Wert)**	**C-Datentyp**
integer	c_int	int
	c_short	short int
	c_long	long int
	c_long_long	long long int
	c_signed_char	signed char, unsigned char
	c_size_t	size_t
	c_int8_t	int8_t
	c_int16_t	int16_t

Fortran-Datentyp	**Benannte** `iso_c_binding`-**Konstante (kind-Wert)**	**C-Datentyp**
	c_int32_t	int32_t
	c_int64_t	int64_t
	c_int_least8_t	int_least8_t
	c_int_least16_t	int_least16_t
	c_int_least32_t	int_least32_t
	c_int_least64_t	int_least64_t
	c_int_fast8_t	int_fast8_t
	c_int_fast16_t	int_fast16_t
	c_int_fast32_t	int_fast32_t
	c_int_fast64_t	int_fast64_t
	c_intmax_t	intmax_t
	c_intptr_t	intptr_t
real	c_float	float
	c_double	double
	c_long_double	long double
complex	c_float_complex	float _Complex
	c_double_complex	double _Complex
	c_long_double_complex	long double _Complex
logical	c_bool	_Bool
character	c_char	char
Quelle: J3/04-007 Fortran 2003 Working Draft		

Das `iso_c_binding`-Modul stellt keine speziellen kind-Werte für `unsigned`-Integer-Datentypen zur Verfügung. Im Bedarfsfall sind die entsprechenden kind-Werte für die `signed`-Datentypen zu verwendet.

Nicht jeder Fortran-Compiler unterstützt alle genannten C-Datentypen und die unterstützten Datentypen können sich com-

pilerspezifisch in der Byteanzahl unterscheiden. Die nächste Tabelle zeigt kurz auf, welche kind-Konstanten derzeit (1. Dez. 2007) von einigen Compilern definiert werden.

	c_int	c_short	c_long	c_long_long	c_signed_char	c_size_t	c_int8_t
g95	4	2	4	8	1	4	1
gfortran	4	2	4	8	1	4	1
ifort	4	2	4	8	1	4	1
f95	4	2	4	8	1	4	
	c_int16_t	c_int32_t	c_int64_t	c_int_least8_t	c_int_least16_t	c_int_least32_t	c_int_least64_t
g95	2	4	8	1	2	4	8
gfortran	2	4	8	1	2	4	8
ifort	2	4	8	1	2	4	8
f95				1	2	4	8
	c_int_fast8_t	c_int_fast16_t	c_int_fast32_t	c_int_fast64_t	c_intmax_t	c_intptr_t	c_float
g95	1	4	4	8	4	4	4
gfortran	(1)	(1)	(1)	(1)	8	(1); 4 oder 8	4
ifort	1	4	4	8	8	4	4
f95	1	2	4	8	8		4

	c_-double	c_-long_-double	c_-float_-complex	c_-double_-complex	c_-long_-double_-complex	c_-bool	c_-char
g95	8	-1	4	8	-1	-1	1
gfortran	8	(1), oft 10	4	8	(1), oft 10	1	1
ifort	8	-1	4	8	-1	1	1
f95	8	-3	4	8	-3	1	1

(1) Der Wert ist systemabhängig

- g95: g95 0.91! Nov 29 2007
- gfortran: GNU Fortran (GCC) 4.3.0 20071201 (experimental)
- ifort: Intel Fortran Compiler 10.1 20070913
- f95: Sun Studio Express, June 2007

positive Zahl	Konstante ist bekannt, C-Datentyp wird unterstützt
negative Zahl	Konstante ist bekannt, C-Datentyp wird nicht unterstützt
	unbekannte Konstante

61.8. „call by value“ vs. „call by reference“

Im EINFÜHRENDEN BEISPIEL[1] wurden die Funktionsargumente „call by value“ übergeben. Das Variablenattribut `value` stellt

1 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN_UND_C%23EIN_EINFACHES_BEISPIEL%20

dieses Verhalten sicher. Wird dieses Attribut nicht gesetzt, so gilt „call by reference“.

Beispiel:

Beispiel funktioniert mit Compiler

- g95 (0.91!) May 10 2007: ja
- gfortran 4.3.0 20070723 (experimental): ja
- Intel Fortran Compiler 10.0: ja
- Sun Studio Express - June 2007: ja

Anmerkungen:

Fortran-Code *bsp.f95*: **Fortran 2003-Code**

```
program bsp
  implicit none

  interface
    subroutine zahl( a, b ) bind( c )
      use, intrinsic :: iso_c_binding
      integer( kind=c_int ), value :: a
      integer( kind=c_int )        :: b
    end subroutine zahl
  end interface

  call zahl(5, 7)
! Ausgabe:
!   Ergebnis = 35
end program bsp
```

C-Code *bsp.c*: **Programmcode**

```
#include <stdio.h>

void zahl(int a, int *b)
{
  printf("%s%d\n", "Ergebnis = ", a * *b);
}
```

61.9. Globale C-Variablen

Muss in Fortran auf globale C-Variablen zugegriffen werden, so sind diese im Gültigkeitsbereich eines Fortran-Modul zu spezifizieren.

61.10. Felder

Interoperabilität zwischen Fortran und C ist nur mit Feldern definierter Größe gegeben. Allozierbare Felder oder Zeigerfelder sind nicht erlaubt.

Beispiel:

Beispiel funktioniert mit Compiler

- g95 (0.91!) May 10 2007: ja
- gfortran 4.3.0 20070723 (experimental): ja
- Intel Fortran Compiler 10.0: ja
- Sun Studio Express - June 2007: ja

Anmerkungen:

Fortran 2003-Code

```
program bsp
  implicit none
  integer, dimension( 3 ) :: a = (/ 1, 2, 3 /)
  interface
    subroutine feld1( f ) bind( c )
      use, intrinsic :: iso_c_binding

      integer( c_int ), dimension(*) :: f
    end subroutine feld1
  end interface
  write (*,*) "Feld a vorher: ", a
  call feld1( a )
  write (*,*) "Feld a nachher: ", a
```

```
! Ausgabe:
!   Feld a vorher:  1 2 3
!   Feld a nachher:  999 2 3
end program bsp
```

Programmcode

```
void feld1(int f[])
{
   f[0] = 999;
}
```

61.11. Übergabe von Zeichenketten

Beispiel:

Beispiel funktioniert mit Compiler

- g95 (0.91!) May 10 2007: ja
- gfortran 4.3.0 20070723 (experimental): nein
- Intel Fortran Compiler 10.0: ja
- Sun Studio Express - June 2007: ja

Anmerkungen:
gfortran lehnt dies ab, da dies ungültiger Fortran-Syntax ist: Fehler: Character argument 'str_in' at (1) must be length 1 because procedure 'string1' is BIND(C)

Fortran-Code *bsp.f95*: **Fortran 2003-Code**

```
program bsp
  use, intrinsic :: iso_c_binding
  implicit none
  interface
    subroutine string1( str_in ) bind( c )
      use, intrinsic :: iso_c_binding

      character( kind=c_char, len=* ) :: str_in ! Ungültiges Fortran
```

```
 2003 da nur len=1 erlaubt ist
    end subroutine string1
  end interface

  call string1( c_char_"Greetings from Fortran" // c_null_char )

! Ausgabe:
!   "Greetings from Fortran"
end program bsp
```

C-Code *bsp.c*: **Programmcode**

```
#include <stdio.h>
void string1(char str_in[])
{
  printf("%s \n", str_in);
}
```

Für len=1 funktioniert das Beispiel mit allen angeführten Compilern. Das `len`-Attribut kann im Übrigen auch weggelassen werden; einige unterstützen als compilerspezifische Erweitungen auch andere Längen. Im Fortran 2003-Working-Draft wird in „Note 15.23“ eine andere Möglichkeit für die Übergabe von Zeichenketten angeführt. Diese folgt im wesentlichen der Annahme:

> C-Zeichenketten sind `char`-Felder mit einem terminierenden `\0`-Zeichen und können deshalb in Fortran als Felder vom Datentyp `character` mit einem kind-Wert `c_char` angesprochen werden.

Daher wird im Interface der Dummy-Parameter in Form eines Feldes aus Zeichen des Typs `c_char` deklariert. Das entspräche auch genau der aufzurufenden C-Funktion. Allerdings ist solcherart verfasster Code dann nicht mit allen Compilern übersetzbar (Stand Juli 2007).

Beispiel:

{{Fortran:Vorlage: Isocbinding|ja|ja|ja|nein|Sun Studio Express-Fehlermeldung:

```
... Zusicherung >>addr<< nicht
erf?llt.
"bsp.f95", Line = 13, Column = 1:
INTERNAL: Interrupt: Abgebrochen}}
```

Fortran-Code *bsp.f95*: **Fortran 2003-Code**

```
program bsp
  use, intrinsic :: iso_c_binding
  implicit none
  interface
    subroutine string1( str_in ) bind( c )
      use, intrinsic :: iso_c_binding

      character( kind=c_char ), dimension(*) :: str_in
    end subroutine string1
  end interface

  call string1( c_char_"Greetings from Fortran" // c_null_char )

! Ausgabe:
!   "Greetings from Fortran"
end program bsp
```

Beispiel:

Damit das obige Beispiel auch mit dem "Sun Studio-Fortrancompiler" funktioniert, darf der String nicht direkt dem Unterprogramm übergeben werden, sondern muss zuvor in einer Variablen abgelegt werden.

Beispiel funktioniert mit Compiler

- g95 (0.91!) May 10 2007: ja
- gfortran 4.3.0 20070723 (experimental): ja
- Intel Fortran Compiler 10.0: ja
- Sun Studio Express - June 2007: ja

Anmerkungen:

Fortran-Code *bsp.f95*: **Fortran 2003-Code**

```
program bsp
  use, intrinsic :: iso_c_binding
  implicit none

  character(len=35) :: str
  interface
    subroutine string1( str_in ) bind( c )
      use, intrinsic :: iso_c_binding

      character( kind=c_char ), dimension(*)  :: str_in
    end subroutine string1
  end interface

  str = "Greetings from Fortran" // c_null_char
  call string1( str )

! Ausgabe:
!   "Greetings from Fortran"
end program bsp
```

Benannte `iso_c_binding`-Konstanten für Zeichen mit spezieller Bedeutung in C:

Benannte Konstante	**Wert**
c_null_char	'\0'
c_alert	'\a'
c_backspace	'\b'
c_form_feed	'\f'
c_new_line	'\n'
c_carriage_return	'\r'
c_horizontal_tab	'\t'
c_vertical_tab	'\v'

61.12. Enumerationen

Mit Fortran 2003 sind auch in dieser Programmiersprache Enumerationen (Aufzählungstypen) möglich. Die Werte in einer solchen Enumeration besitzen einen `integer`-Datentyp. Der `kind`-Wert ist nicht festgelegt, wird jedoch so gewählt, dass im Rahmen der jeweiligen Möglichkeiten alle Enumeratoren erfasst sind.

Die von C-Enumerationen bekannten Eigenschaften gelten gleichermaßen für Fortran-Enumerationen, z.B.:

- ohne explizite Zuweisung von Werten wird mit dem Wert 0 gestartet.
- ohne explizite Zuweisung von Werten wird in der Anordnungsreihenfolge der Elemente sukzessiv immer um 1 hochgezählt.
- Wurde dem Vorgängerelement eine Ganzzahl zugewiesen, dem Element jedoch nicht, so ist der Wert dieses Elementes die dem Vorgängerelement zugeordnete Ganzzahl + 1.

Beispiel:

Beispiel funktioniert mit Compiler

- g95 (0.91!) May 10 2007: ja
- gfortran 4.3.0 20070723 (experimental): ja
- Intel Fortran Compiler 10.0: nein
- Sun Studio Express - June 2007: nein

Anmerkungen:

> Sun-Fortran-Compiler und Intel-Fortran-Compiler unterstützen momentan noch keine Enumerationen.

Fortran-Code *bsp.f95*: **Fortran 2003-Code**

```
program bsp
  implicit none
  enum, bind(c)
    enumerator :: MO=1, DI=2, MI=3, DO=4, FR=5, SA=6, SO=7
  end enum

  interface
    subroutine tag( w ) bind( c )
      use, intrinsic :: iso_c_binding
      integer( c_int ), value :: w
    end subroutine tag
  end interface

  call tag(MI);
! Ausgabe:
!   Mittwoch
end program bsp
```

C-Code *bsp.c*: **Programmcode**

```
#include <stdio.h>
typedef enum {
  MO=1, DI=2, MI=3, DO=4, FR=5, SA=6, SO=7
} wochentag;
void tag(wochentag w)
{
  switch(w)
  {
    case MO:
      printf("Montag\n");
      break;
    case DI:
      printf("Dienstag\n");
      break;
    case MI:
      printf("Mittwoch\n");
      break;
    case DO:
      printf("Donnerstag\n");
      break;
    case FR:
      printf("Freitag\n");
      break;
    case SA:
      printf("Samstag\n");
```

```
      break;
    case SO:
      printf("Sonntag\n");
      break;
    default:
      printf("Kein Tag\n");
  }
}
```

61.13. Zeiger

Für das C-Zeiger-Handling stellt Fortran 2003 im `iso_c_binding`-Modul den Datenverbund `c_ptr` und einige Unterprogramme bereit.

UP	Beschreibung
l = **c_associated** (c_ptr1 [, c_ptr2])	Prüft den Assoziationsstatus von c_ptr1. Diese Funktion ermittelt also, ob c_ptr1 überhaupt assoziert ist, oder ob c_ptr1 mit c_ptr2 assoziert ist.
c_ptr = **c_loc** (x)	Gibt die Adresse von x zurück.
c_f_pointer (c_ptr, fptr [, shape])	Wandelt einen C-Zeiger `c_ptr` in einen Fortran-Zeiger `fptr` um. Die Vorgabe von `shape` ist nur dann erforderlich und möglich, wenn `fptr` ein Feld ist. `fptr` ist `intent( inout )`.

Beispiel:

Beispiel funktioniert mit Compiler

- g95 (0.91!) May 10 2007: ja

- gfortran 4.3.0 20070723 (experimental): ja
- Intel Fortran Compiler 10.0: ja
- Sun Studio Express - June 2007: ja

Anmerkungen:

Sun-Linker-Warnmeldung:

```
Warning: alignment 4 of symbol
'ptr1', ... in bsp_c.o is smaller
than 16 in bsp_f90.o
```

Intel-Linker-Warnmeldung:

```
Warning: alignment 4 of symbol
'ptr1', ... in bsp_c.o is smaller
than 8 in bsp_f90.o
```

Fortran-Code *bsp.f95*: **Fortran 2003-Code**

```
module cglob
  use, intrinsic :: iso_c_binding
  type( c_ptr ), bind( c )                :: ptr1, ptr2, ptr3
  real( kind=c_float ), target, bind( c ) :: a, b
end module cglob
program bsp
  use cglob
  implicit none

  real, pointer :: p => null()

! *** Zuordnungsstatus ***
  write (*,*) "Ist ptr1 assoziert? ", c_associated( ptr1 )
  write (*,*) "Ist ptr2 assoziert? ", c_associated( ptr2 )
  write (*,*) "Ist ptr2 mit ptr3 assoziert? ", c_associated( ptr2,
 ptr3 )
  write (*,*) "Ist ptr2 mit &a assoziert? ", c_associated( ptr2,
 c_loc(a) )

! *** Wert von ptr2? ***
  call c_f_pointer( ptr2, p )
```

```
  write (*,*) "Wert von ptr2? ", p
! *** ptr1 neu setzen ***
  ptr1 = c_loc( b )
! *** Wert von ptr1 ***
  call c_f_pointer( ptr1, p )
  write (*,*) "Wert von ptr1? ", p

! Ausgabe:
!   Ist ptr1 assoziert?  F
!   Ist ptr2 assoziert?  T
!   Ist ptr2 mit ptr3 assoziert?  F
!   Ist ptr2 mit &a assoziert?  T
!   Wert von ptr2 (bzw. p)?  5555.66
!   Wert von ptr1 (bzw. p)?  -12.3
end program bsp
```

C-Code *bsp.c*: **Programmcode**

```
float a = 5555.66;
float b = -12.3;
float *ptr1 = 0;
float *ptr2 = &a;
float *ptr3 = &b;
```

Nun ist die Zeiger-Verwendung wie im obigen Beispiel skizziert, eher die Ausnahme als die Regel. Wesentlich häufiger trifft man Zeiger in C-Bibliotheken im Zusammenhang mit Funktionen an. Dort dienen sie aus Anwendersicht als `return`-Werte oder der Parameterübergabe „call-by-reference“. Oft sind dabei auch Zeiger auf Strukuren im Spiel. Näheres dazu folgt im nächsten Abschnitt.

Für C-Zeiger auf den Wert `NULL` bietet das ISO-C-Binding-Modul die Konstante `C_NULL_PTR`.

61.14. Datenverbund

Die einzelnen Datenelemente der Struktur / des Datenverbundes müssen in Fortran und C selbstverständlich äquivalenten Datentyp aufweisen. Die Bezeichnungen der jeweiligen Datenele-

mente müssen nicht beibehalten werden. Sehr wohl müssen aber die Positionen der Einzelelemente im Fortran-Datenverbund der nachzubildenden C-Struktur entsprechen.

Für die Gewährleistung der Interoperabilität darf ein an C gebundener Datenverbund in Fortran (`struct`) **keine** Zeiger mit dem `pointer`-Attribut oder allozierbaren Felder als Datenelemente enthalten. Sind in der C-Struktur Zeiger vorhanden, so sind diese im entsprechenden Fortran-Datenverbund mittels `type( c_ptr )` zu beschreiben.

Für Bitfelder oder Unions gibt es in Fortran keine entsprechenden Gegenstücke.

Beispiel:

Beispiel funktioniert mit Compiler

- g95 (0.91!) May 10 2007: ja
- gfortran 4.3.0 20070723 (experimental): ja
- Intel Fortran Compiler 10.0: ja
- Sun Studio Express - June 2007: ja

Anmerkungen:
gfortran: Im Gegensatz zu den anderen Compilern müssen bei der Deklaration der Subroutine `print_v` im Interface die Klammern für die leere Parameterliste vor dem `bind(C)` zwingend angestschrieben werden, ansonsten Fehlermeldung beim Compilieren; diese Klammer ist im Fortran 2003 Standard vorgeschrieben.

Fortran 2003-Code

```
module cglob
  use, intrinsic :: iso_c_binding
  type, bind( c ) :: verbund
    integer( kind=c_int ) :: a
    real( kind=c_float )  :: b
  end type verbund
  type(verbund), bind( c ) :: v
```

```
end module cglob
program bsp
  use cglob
  implicit none
  interface
    subroutine set_v( a_in, b_in) bind( c )
      use, intrinsic :: iso_c_binding

      integer( c_int ), value :: a_in
      real( c_float ), value :: b_in
    end subroutine set_v

    subroutine print_v() bind( c )
    end subroutine print_v

  end interface
  call set_v( 5, -10.9)
  call print_v()
  write (*,*) "Fortran-Ausgabe: ", v
  v%a =  99
  call print_v()
  write (*,*) "Fortran-Ausgabe: ", v
! Ausgabe:
!   C-Ausgabe: 5  -10.900000
!    Fortran-Ausgabe:  5 -10.9
!   C-Ausgabe: 99  -10.900000
!    Fortran-Ausgabe:  99 -10.9
end program bsp
```

Programmcode

```
#include <stdio.h>
typedef struct {
  int a;
  float b;
} verbund;
verbund v;
void set_v(int a_in, float b_in)
{
  v.a = a_in;
  v.b = b_in;
}
void print_v()
{
  printf("C-Ausgabe: %i  %f\n", v.a, v.b);
}
```

Beispiel: Struktur als Rückgabewert und Argument einer Funktion - Teil 1

Bekannt seien zwei Funktionsprototypen in der Programmiersprache C:

```
Xyz *get_xyz();
```

und

```
void set_xyz(Xyz *x);
```

Xyz sei ein C-`struct`, dessen Inhalt hier nicht näher interessiert.

Die Schnittstelle, mit dem in Fortran die C-Anbindung der Funktionen realisiert wird, könnte so aussehen:

```
interface
  function get_xyz() bind( c )
    use, intrinsic :: iso_c_binding
    type( c_ptr )  :: get_xyz
  end function get_xyz

  subroutine set_xyz( x ) bind( c )
    use, intrinsic :: iso_c_binding
    type( c_ptr ), value  :: x
  end subroutine set_xyz
end interface
```

Und schon können diese Funktionen auch in Fortranroutinen direkt Anwendung finden, z.B:

```
type( c_ptr ) :: x56

x56 = get_xyz()
```

```
call set_xyz( x56 )
```

Wie man anhand dieses Beispiels erkennt, ist die interne Struktur von `Xyz` in diesem Fall vollkommen irrelevant. Es ist nicht nötig, in der API-Dokumentation oder in den C-Headerdateien nachzuforschen, wie der C-`struct` konkret aufgebaut ist. Für den Fortran-Programmierer ist dies immer ein `type( c_ptr )` (natürlich nur, solange Zeiger auf Strukturen gefordert sind). Wird ein Zeiger des Typs `c_ptr` nur via Fortranprogramm zwischen verschiedenen C-Funktionen übergeben oder zurückgeliefert, so ist auch keine Umwandlung in einen Fortran-Zeiger erforderlich und wäre mangels genauer Kenntnis des Aufbaus von `Xyz` hier auch nicht möglich. Erst dann, wenn von Fortran aus auf einzelne Elemente eines C-`struct` zugegriffen werden soll oder mit Kopien der Struktur hantiert wird, muss diese Struktur in Fortran nachgebaut werden.

Beispiel: Struktur als Rückgabewert und Argument einer Funktion - Teil 2

Beispiel funktioniert mit Compiler

- g95 (0.91!) May 10 2007: ja
- gfortran 4.3.0 20070723 (experimental): ja
- Intel Fortran Compiler 10.0: ja
- Sun Studio Express - June 2007: ja

Anmerkungen:

Fortran 2003-Code

```
module test
  use, intrinsic :: iso_c_binding
  implicit none
```

```
  type, bind( c ) :: A
  ! Das funktioniert nicht mit allen Kompilern; versuchen Sie
 ansonsten:
  !   type A
  !   sequence
    integer( c_int ) :: xc, yc
    type( c_ptr )    :: str
  end type
  interface
    function get_a() bind( c )
      use, intrinsic :: iso_c_binding
      type( c_ptr ) :: get_a
    end function get_a
    subroutine print_a( x ) bind( c )
      use, intrinsic :: iso_c_binding
      type( c_ptr ), value :: x
    end subroutine print_a
  end interface
end module test
program main
  use test
  implicit none
  type( c_ptr )                :: x
  type( A ), pointer           :: fptr
  character( len=9 ), pointer :: strptr
  x = get_a()

! C-Ausgabe
  call print_a( x )
!  Ausgabe:
!    x = 5
!    y = 997
!    str = Irgendwas
! Fortran-Ausgabe
  call c_f_pointer( x, fptr)
  call c_f_pointer( fptr%str, strptr )  ! <--- Fehlerquelle bei
 gfortran
  write(*,*) fptr%xc, fptr%yc, strptr
!  Ausgabe
!    5 997 Irgendwas
end program main
```

Programmcode

```
#include <stdlib.h>
#include <stdio.h>
typedef struct
```

```
{
  int x;
  int y;
  const char *str;
} A;
A *get_a()
{
  A *a = ( A * )malloc( sizeof( A ) );

  a->x = 5;
  a->y = 997;
  a->str = "Irgendwas";
  return a;
}
void print_a( A *v )
{
  printf("x = %d\n", v->x);
  printf("y = %d\n", v->y);
  printf("str = %s\n", v->str);
}
```

Bei der Umwandlung des C-String-Zeigers in einen Fortran-Zeiger ist die genaue Stringlänge erforderlich. Ist diese zu klein gewählt, so ist nur ein Teil des C-Strings über den Fortran-Zeiger sichtbar. Wird diese Stringlänge zu groß gewählt, dann wird diese Länge auch voll ausgenutzt und nach dem eigentlich gewünschten String folgen noch ein Menge x-beliebige Zeichen, da in Fortran der C-String-Begrenzer `\0` nicht die Bedeutung wie in C besitzt.

Im obigen Beispiel wurde die Speicherplatzreservierung und Wertebelegung für eine Variable des Typs `A` in einer C-Funktion erledigt. Nun soll diese Aufgabe im Fortran-Programm wahrgenommen werden und dann diese in Fortran mit Werten belegte Variable an die `print_a`-Funktion übergeben werden. Kein Problem, möchte man im ersten Augenblick meinen. Doch der `c_ptr`-Typ im Datenverbund für den Zeiger auf eine Zeichenkette macht Probleme. Der g95-Compiler duldet bspw. in der `c_loc`-Funktion keine Zeichenketten mit einer Länge größer als 1. Der Compiler "castet" auch nicht von selbst im Datenverbundkonstruktor eine Zeichenkette in den Typ `c_ptr`. Eine Möglichkeit,

diese Probleme zu umgehen, besteht darin, einfach einer C-Funktion einen String zu übergeben und die Adresse dieses Strings zurückgeben zu lassen. Diese Variante wird auch im folgenden Beispiel verwendet.

Beispiel: Struktur als Rückgabewert und Argument einer Funktion - Teil 3

Beispiel funktioniert mit Compiler

- g95 (0.91!) May 10 2007: ja
- gfortran 4.3.0 20070723 (experimental): ja
- Intel Fortran Compiler 10.0: ja
- Sun Studio Express - June 2007: ja

Anmerkungen:

Fortran 2003-Code

```
module test
  use, intrinsic :: iso_c_binding
  implicit none

  type, bind( C ) :: A
    integer( c_int ) :: xc, yc
    type( c_ptr )    :: str
  end type
  type( A ), target  :: a1          ! bei Verwendung des g95 könnte
 diese Var.deklaration
                                    ! auch im Hauptprogramm als lokale
 Variable
                    ! vorgenommen werden. Der Sun-Compiler erlaubt
                    ! kein gleichzeitiges C-Binding mit einer lokalen
 Var.
  interface
    subroutine print_a( x ) bind( c )
      use, intrinsic :: iso_c_binding
      type( c_ptr ), value :: x
    end subroutine print_a

    function c_string_addr( s ) bind( c )
      use, intrinsic :: iso_c_binding
      character( c_char ) :: s
```

```
      type( c_ptr )       :: c_string_addr
    end function c_string_addr
  end interface
end module test
program main
  use test
  implicit none
  character( len = 30 ) :: str

  str =  "Das ist ein Beispiel" // C_NULL_CHAR
  a1 = A( 14, 56, c_string_addr( str ) )

  call print_a( c_loc( a1 ) )
!  Ausgabe:
!    x = 14
!    y = 56
!    str = Das ist ein Beispiel
end program main
```

Programmcode

```
#include <stdlib.h>
#include <stdio.h>
typedef struct
{
  int x;
  int y;
  const char *str;
} A;
void print_a( A *v )
{
  printf("x = %d\n", v->x);
  printf("y = %d\n", v->y);
  printf("str = %s\n", v->str);
}
char *c_string_addr( char *str )
{
  return str;
}
```

61.15. Problematische Kamelhöckerschreibweise?

C ist case-sensitive, Fortran nicht. Oft sind Funktionen in C-Bibliotheken in der KAMELHÖCKERSCHREIBWEISE[2] vorzufinden, z.B.

```
void writeHallo();
```

Die Einbindung dieser C-Funktion in ein Fortran-Programm mit

```
 interface
   subroutine writeHallo() bind(c)
   end subroutine writeHallo
 end interface
```

könnte beim Linken eine Fehlermeldung der Art

```
undefined reference to 'writehallo'
```

liefern.

Was ist passiert? Wie schon angedeutet, ist Fortran case-insensitive. Für einen Fortran-Compiler ist `writeHallo` gleich `writehallo` oder `WRITEHaLLO`. Für einen C-Compiler wären das alles unterschiedliche Funktionsbezeichner. Also wandelt der Fortran-Compiler die "unnütze" Groß-/Kleinschreibung in eine einheitliche Schreibweise um und der C-Compiler nicht. Dementsprechend findet der Linker auch keine Funktion `writehallo`. Es gibt eben nur die nicht-idente C-Funktion `writeHallo`.

2 HTTP://DE.WIKIPEDIA.ORG/WIKI/BINNENMAJUSKEL

Aber auch für dieses Problem gibt es in Fortran eine Lösung. Mit dem `bind`-Attribut können zusätzlich auch noch Labels (Benennungen) vergeben werden, z.B.:

```
bind( c, name="xyz" )
```

Solche Labels in Stringform sind nicht von der "Kopf ab"-Strategie des Fortran-Compilers betroffen. Das ist keine C-Binding-Spezialität, sondern trifft generell für alle Zeichenketten zu. Ein

```
interface
  subroutine writeHallo() bind(c, name="writeHallo")
  end subroutine writeHallo
end interface
```

sollte das beschriebene Link-Problem lösen.

61.16. Sonstiges

- Auch C-Funktionszeiger kennt das `iso_c_binding`-Modul. Zu diesem Zwecke gibt es den Datenverbund `c_funptr`, einige Umwandlungsfunktionen ähnlich jenen im Abschnitt ZEIGER[3] behandelten und die Konstante `C_NULL_FUNPTR`.

3 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20FORTRAN%20UND%20C%23ZEIGER%20

62. Fortran und Python

F2PY

F2py (Fortran to Python interface generator) ist ein Wrapper, der Fortran-Module, -Subroutinen oder -Funktionen in sogenannte SharedObject-Dateien kompiliert, die dann in Python als normales Modul importiert werden können. Einsatzgebiete sind vor allem numerische Berechnungen, in denen sowohl die Geschwindigkeit der Fortanroutinen, aber auch die Flexibilität von Pythonscripten genutzt werden soll.

62.1. Voraussetzungen

Erforderlich für die Installation von f2py sind einerseits eine Python-Version ab 2.0 zusammen mit den Erweiterungsmodulen NumPy und SciPy und außerdem ein Fortran-Compiler. Genaue Informationen finden Sie auf der Homepage des f2py-Projekts, die unten angegeben ist. Es ist sehr wichtig, dass Sie zunächst alle Komponenten installiert haben, bevor Sie fortfahren. Die Erfahrung hat gezeigt, dass dies manchmal gar nicht so einfach ist. Um zu überprüfen, ob Ihr f2py funktionsfähig ist, tippen Sie einfach *f2py* in die Konsole ein. Wünschenswerterweise sollte dies einige Informationen über das Programm auf Ihrem Bildschirm anzeigen.

62.2. Beispiel

Als Testbeispiel wählen wir ein kleines Programm namens punktrechnung.f90. Es enthält ein Modul und eine Subroutine zur Berechnung von Division und Multiplikation. Wir wollen diese Routinen später von Python aus aufrufen können und die Funktionalitäten nutzen.

62.2.1. Fortran Code

Fortran 90/95-Code (free source form)

```
!File: punktrechnung.f90
    module punktrechnung
      real,parameter::pi=3.14

    contains

    subroutine division(x,y)
        integer::x,y
        print*, "x/y = ",x/y
    end subroutine division

    end module punktrechnung

    subroutine multi(x,y)
        integer::x,y
        print*, "x*y= ",x*y
    end subroutine multi
```

62.2.2. Wrappen

Mit dem Befehl: *f2py -c -m meinmodul punktrechnung.f90*

Erstellen wir eine Objekt-Datei mit der Endung .so, die von Python als Modul importiert werden kann. Das Modul trägt den Namen "meinmodul" und es beinhaltet alle Funktionalitäten der Datei punktrechnung.f90

62.2.3. Zugriff

Im interaktiven Modus von Python können wir dieses Modul nun importieren und die Funktionen nutzen:

```
>>> import meinmodul
>>> meinmodul.multi(5,4)
 x*y=           20
>>> meinmodul.punktrechnung.diff(36,6)
 x/y =            6
```

62.3. Weblinks

- OFFIZIELLE SITE DES F2PY-PROJEKTS[1]

1 HTTP://CENS.IOC.EE/PROJECTS/F2PY2E/

Teil VIII.

Fortran-„Dialekte“

63. F

63.1. Was ist F?

F ist ein Fortran 90/95-Subset. Das bedeutet auch, dass sich F-Programmcode immer mit Fortran 90/95-Compilern übersetzen lassen sollte. Umgekehrt ist das natürlich oft nicht möglich. Grob gesagt, wurde bei F auf die bedingungslose Kompatibilität mit FORTRAN 77-Code gepfiffen und nur die moderneren Fortran 90/95-Sprachmittel erlaubt. Die überbordenden Möglichkeiten und Schreibweisen zur Erreichung ein und desselben Ziels wurden also deutlich eingebremst. F ist somit ein bisschen leichter erlernbar als Fortran 90/95 in seiner Gesamtheit und führt durch restriktivere Regeln automatisch zu einem etwas stringenteren Programmierstil. Und das war bei der Einführung von F auch das Ziel: Einsatz von F in der Ausbildung und Lehre, wenngleich natürlich von den F-Schöpfern der Einsatz in praktischen Projekten auch gerne gesehen worden wäre.

Ein expliziter F-Compiler ist auf [ftp://ftp.swcp.com/pub/walt/F ftp.swcp.com/pub/walt/F unter FortranTools] für die Betriebssysteme MS Windows und Linux zu finden. In diesen Softwarepaketen sind unter anderem das F-Manual und etliche Beispiele in der F-Programmiersprache enthalten.

63.2. Einige Unterschiede zu Fortran 90/95

Im Nachfolgenden werden Fortran 90/95- und F-Beispiele gegenüber gestellt. Das soll aber nicht bedeuten, dass die Fortran 90/95-Beispiele zwingend so formuliert werden müssen oder optimaler Fortran 90/95-Code sind. Es soll nur gezeigt werden, welche Freiheiten Fortran 90/95 im Gegensatz zu F erlaubt. F ist ein Subset von Fortran 90/95 und die als F-Code deklarierten Beispiele enthalten somit auch immer gültigen Fortran 90/95-Code.

63.2.1. Diverses

- Strengere Regeln bei `end`-Statements für Hauptprogramm, Unterprogramme und Module:

Fortran 90/95	**F**
program bsp ! ... end	program bsp ! ... end program bsp

63.2.2. Datentypen

- In F gibt es den Datentyp `double precision` nicht. Er kann aber problemlos wie in Fortran 90 /95 mittels `real` und `kind`-Attribut ersetzt werden.
- Die alte (FORTRAN 77)-Form ohne Doppel-Doppelpunkt zwecks Variablendeklaration ist in F nicht erlaubt:

Fortran 90/95	**F**
program bsp real r r = 12.5 ! ... end program bsp	program bsp real :: r r = 12.5 ! ... end program bsp

- In F gibt es keine impliziten Datentypvereinbarungen. Der Datentyp muss immer explizit festgelegt werden:

Fortran 90/95	**F**
program bsp i = 12 ! ... end program bsp	program bsp integer :: i = 12 ! ... end program bsp

63.2.3. Ein- / Ausgabe

- Die vereinfachte Schreibweise `write( *, * )` und `read( *, * )` funktioniert bei F nicht. F akzeptiert Apostrophe nicht als Zeichenkettenbegrenzer:

Fortran 90/95	**F**
program bsp write(*, *) 'Hallo Welt!' ! Alternativ auch: print *, 'Hallo Welt!' end program bsp	program bsp write(unit = *, fmt = *) "Hallo Welt!" ! Alternativ auch: print *, "Hallo Welt!" end program bsp

- Kein `format`, keine Marken:

Fortran 90/95	**F**
program bsp real :: a = 30.0 write(*, fmt = 999) a 999 format(F10.3) end program bsp	program bsp real :: a = 30.0 write(unit = *, fmt = "(F10.3)") a end program bsp

63.2.4. Schleifen

- F kennt keine `do-while`-Schleife.

63.2.5. Funktionen und Subroutinen

- Egal, ob die Parameterliste leer ist oder nicht, in F müssen immer die Klammern im Anschluss an den Unterprogrammnamen geschrieben werden
- Externe Unterprogramme sind nicht erlaubt.

Fortran 90/95	F
program bsp call schreibe end program bsp subroutine schreibe write(*, *) "Hallo" end subroutine schreibe	program bsp call schreibe() contains subroutine schreibe() write(unit = *, fmt = *) "Hallo" end subroutine schreibe end program bsp

63.2.6. Module

F ist restriktiver bei der Verwendung von Modulen als Fortran 90/95. Für Modulvariablen und -unterprogramme muss immer ein Zugriffspezifizierer `public` oder `private` vorgegeben werden.

Fortran 90/95	F
module modu real :: pi = 3.1416 contains subroutine schreibe() write(*, *) "Hallo" end subroutine schreibe end module modu program bsp use modu write(*, *) pi call schreibe() end program bsp	module modu private real, public :: pi = 3.1416 public :: schreibe contains subroutine schreibe() write(unit = *, fmt = *) "Hallo" end subroutine schreibe end module modu program bsp use modu write(unit = *, fmt = *) pi call schreibe() end program bsp

Weitere Unterschiede sind mittels angegebener Weblinks auffindbar bzw. dem F-Manual zu entnehmen.

63.3. Weblinks

- F Programming Language Homepage[1]
- Introduction to F[2]
- Neue Fortran-Sprache F[3]
- The Fortran Journal Volume 8, Number 6, 1996 November/December[4]

1 http://www.fortran.com/F/
2 http://sip.clarku.edu/tutorials/F.html
3 http://www.hrz.uni-{}wuppertal.de/infos/hrz-{}info/hrz-{}info-{}9607/node15.html
4 http://www.fortran.com/fortran/FJ/9611/

64. Ratfor

64.1. Allgemeines

Ratfor (**Rat**ional **For**tran) ist zum Großteil eine Mischung aus C und Fortran. Es entstand aus der Begeisterung Brian W. Kernighans[1] für die Programmiersprache C. In einer Zeit, als C seinen Siegeszug antrat und Programmierer mit den zum Teil noch archaischen Sprachmerkmalen von FORTRAN 66 und 77 konfrontiert waren, hatte er die fixe Idee, dass auch Fortran-Programmierer mit der Syntax von C beglückt werden müssen. Ratfor-Code wurde dann per Präprozessor in Fortran-Code umgewandelt, um danach den Gang alles Fortran-irdischen zu gehen und per schnödem Fortran-Compiler in Maschinencode übersetzt zu werden. Ratfor dürfte sich einer gewissen Popularität erfreut haben. Darauf deutet auch hin, dass sogar für Fortran 90 ein entsprechender Ratfor-Präprozessor in Form eines Perl-Skripts geschaffen wurde. Gerüchten zufolge wurde in den '80er Jahren des vergangenen Jahrhunderts als möglicher Nachfolger von Ratfor ein Ratfiv [2] realisiert.

Hier nun einige der wesentlichen Ratfor-Sprachmerkmale:

- Die fixe Zeilenform aus FORTRAN 77 entfällt

1 BRIAN W. KERNIGHAN ^{HTTP://DE.WIKIPEDIA.ORG/WIKI/BRIAN%20W.%20KERNIGHAN}

2 EN:RATFIV ^{HTTP://DE.WIKIPEDIA.ORG/WIKI/EN%3ARATFIV}

- Anweisungen werden mit Strichpunkt abgeschlossen
- Blöcke werden in geschweifte Klammern eingefasst
- `for`-Schleifen, `if`-Verzweigungen, etc. nach Art der Programmiersprache C
- `switch`-Verzweigung (nur Ratfor 77, ohne `break`)
- Kommentare werden mit # eingeleitet
- `<, <=, ..., !=` anstelle von `.LT., .LE., ..., .NE.`
- etc.

Nachfolgend je ein kleineres Beispiel, um einen kurzen Einblick in Ratfor 77 und Ratfor 90 und die aus diesen Ratfor-Codes per Präprozessor generierten Fortran-Codes zu geben.

64.2. Ratfor 77

Ratfor	**FORTRAN 77**
program bsp i = 2 switch(i){ case 1: write(*, *) "Fall 1"; case 2: write(*, *) "Fall 2" default: write(*, *) "Default" } end	C Output from Public domain Ratfor, version 1.0 program bsp i = 2 I23000=(i) goto 23000 23002 continue write(*, *) "Fall 1" goto 23001 23003 continue write(*, *) "Fall 2" goto 23001 23004 continue write(*, *) "Default" goto 23001 23000 continue if (I23000.eq.1)goto 23002 if (I23000.eq.2)goto 23003 goto 23004 23001 continue end

64.3. Ratfor 90

Ratfor	Fortran 90
• Rator 90 - Beispiel program bsp integer :: i real :: x = 10.5 if(x != 10.0) { for(i = 0; i < 3; i = i + 1) { write(*,*) 'IF Hallo ', i; } } else { write(*,*) 'ELSE Hallo'; } end program bsp	! Rator 90 - Beispiel program bsp implicit none integer :: i real :: x = 10.5 if (x .ne. 10.0) then i=0 do if(.not. (i<3)) then exit end if write(*,*) 'IF Hallo ', i i=i+1 end do else write(*,*) 'ELSE Hallo' end if end program bsp

64.4. Weblinks

- RATFOR 77[3]
- RATFOR 90[4]
- BRIAN W. KERNIGHAN: RATFOR -- A PREPROCESSOR FOR A RATIONAL FORTRAN[5]

<references />

3 HTTP://SEPWWW.STANFORD.EDU/SOFTWARE/RATFOR.HTML
4 HTTP://SEPWWW.STANFORD.EDU/SOFTWARE/RATFOR90.HTML
5 HTTP://WOLFRAM.SCHNEIDER.ORG/BSD/7THEDMANVOL2/RATFOR/RATFOR.HTML

Teil IX.

Sonstiges

65. Cray Pointer

65.1. Einleitung

Cray Pointer (auch *Integer Pointer* genannt) waren und sind teilweise heute noch in Fortran ein Ersatz für die aus der Programmiersprache C bekannten Zeiger. Sie sind nicht Teil der Fortran-Standards, werden jedoch von einigen Compilerherstellern als Erweiterung des Fortran-Sprachumfangs angeboten. *Cray Pointer* haben nicht viel mit den unter Fortran 90/95 neu eingeführten Zeigern gemeinsam und können Portabilitätsprobleme bereiten. Die Bezeichnung *Cray Pointer* deutet auf den "Erfinder" dieser Erweiterung hin - *Cray Research*[1] (firmiert aktuell als *Cray Inc.*).

65.2. Grundlagen

65.2.1. Pärchenbildung: Speicheradresse ⇔ Objekt

```
pointer( pointeradr, pointee ) [, ...]
```

- `pointeradr` ... `integer`-Skalar (Speicheradresse)
- `pointee` ... Skalar oder Vektor

1 CRAY ^{HTTP://DE.WIKIPEDIA.ORG/WIKI/CRAY}

Der Datentyp für `pointeradr` muss groß genug gewählt werden, damit die Speicheradresse auch aufgenommen werden kann. Dies stellt schon das erste Problem dar, da diese "Zeigergröße" systemabhängig ist. In weitere Folge muss dieser Speicherplatzbedarf je Zeiger auch bei der Verwendung von "zeigerarithmetischen Kunststücken" beachtet werden. Um portabel zu bleiben darf `pointeradr` gar nicht deklariert werden. Das Codeschnipsel

```
! ...
integer :: ptradr
pointer( ptradr, var )
! ...
```

würde von *gfortran* und *ifort* auf 32bit-Systemen akzeptieren werden. Der Sun-Compiler weist diesen Code jedoch in jedem Fall mit einer eindeutigen Fehlermeldung ab.

65.2.2. Speicheradresse ermitteln

```
i = loc( var )
```

Diese Funktion gibt die Speicheradresse von `var` zurück.

65.2.3. Speicherplatz anfordern

```
ptradr = malloc( bytes )
```

Fordert `bytes` Speicherplatz an und gibt die erste dazugehörende Speicheradresse zurück.

65.2.4. Speicherplatz freigeben

```
free( ptradr )
```

65.2.5. Sonstiges

- *Cray Pointer* können auch auf Unterprogramme zeigen.

65.3. Beispiel

Fortran 90/95-Code (free source form)

```
program bsp
  implicit none

  real, dimension(4) :: arr = (/ 12.5, -3.3, -55.0, -144.9 /)
  real               :: pointee

! ***** Pärchenbildung und loc() *****
  ! iptradr wird nicht explizit deklariert, der Compiler ermittelt
 automatisch
  ! den erforderlichen Datentyp (kind-Wert) -> 4 Byte-Integer bei
  ! 32bit-Systemen, 8 Byte-Integer bei 64bit-Systemen, ...
  pointer( iptradr, pointee(2, 2) )

  iptradr = loc( arr(1) )
  write( *,* ) pointee
! Ausgabe:
!    12.5      -3.3      -55.0     -144.9
  write( *,* ) pointee(:, 2)
! Ausgabe:
!    -55.0      -144.9
  iptradr = loc ( arr(2) )
  write( *,* ) pointee
! Ausgabe:
!    -3.3      -55.0      -144.9      1.4012985E-45
! ***** Zeigerarithmetik, Ergebnis systemabhängig *****
  iptradr = iptradr + 4      ! + 4 Bytes
  write( *,* ) pointee(1, 1)
! Ausgabe:
!    -55.0
```

```
! ***** malloc() und free() *****
  iptradr = malloc( 16 )
  pointee( 1, 1 ) = 99.9
  pointee( 2, 1 ) = 999.9

  write( *,* ) pointee
! Ausgabe:
!    99.9      999.9      0.0E+0      0.0E+0
  call free( iptradr )
end program bsp
```

Kompilieren, Linken:

- gfortran: `gfortran -fcray-pointer bsp.f90`
- Sun: `f95 bsp.f90`
- Intel: `ifort bsp.f90`

65.4. Weblinks

- GFORTRAN - CRAY POINTERS[2]
- COMPARING POINTERS IN CRAY FORTRAN AND FORTRAN 90[3]
- XLF: INTEGER-POINTER ("CRI-POINTER", "CRAY-POINTER")[4]

<references />

2 HTTP://GCC.GNU.ORG/ONLINEDOCS/GFORTRAN/CRAY-{}POINTERS.HTML

3 HTTP://WWW.CISL.UCAR.EDU/TCG/CONSWEB/FORTRAN90/SCNPOINT.HTML

4 HTTP://WWW.UNICS.UNI-{}HANNOVER.DE/RRZN/GEHRKE/CRI-{}POINTER.HTM

Teil X.

Anhang

66. Eine Gegenüberstellung von grundlegenden Fortran- und C-Sprachelementen

Nachfolgend werden einige grundlegende Sprachelemente von Fortran95 und C tabellarisch gegenübergestellt. Die Gegenüberstellung erfolgt nur schematisch, da hier nur ein Überblick über die Gemeinsamkeiten und Unterschiede der beiden Programmiersprachen aufgezeigt werden soll.

Tabellenlegende:

- ----- : nicht vorhanden
- ... : nicht näher bestimmt, aber irgendwas Sinnvolles und Zulässiges
- *kursiv* Geschriebenes steht für beliebige (aber natürlich nur sinnvolle und zulässige) Variablennamen, Zahlen, etc.

67. Programmgrundstruktur

67.1. Groß-, Kleinschreibung

Fortran95	**C**
Fortran ist case-insensitiv	C ist case-sensitiv

67.2. Kommentare

Fortran95	**C**
! xyz	/* xyz */

67.3. Das Ende einer Anweisung

Fortran95

- Jede Anweisung steht grundsätzlich in einer eigenen Zeile.
- Die Zeilenlänge ist begrenzt.
- Mehrere Anweisungen dürfen auch in einer Zeile stehen, wenn sie durch Semikolon ; voreinander getrennt sind.
- Eine Zeilenfortsetzung muss durch ein Kaufmanns-Und & am Ende der fortzusetzenden Zeile angezeigt werden.
- Die Anzahl der erlaubten Fortsetzungzeilen ist begrenzt.

C

- Einzelne Anweisungungen sind immer durch Semikolon ; zu trennen.
- Eine Anweisung kann sich über mehrere Zeilen erstrecken.

67.4. Hauptprogramm

Fortran95

```
program bsp ... end program bsp
```

C

```
int main(int argc, char **argv)
{ ... }
```

67.5. Unterprogramme

Fortran95

```
call sub1(...)
...
subroutine sub1(...) ... end
subroutine sub1
var = fun1(...)
...
function fun1(...) datentyp ::
fun1 ... fun1 = ... end function
fun1
```

- Prinzipiell werden Parameter per "call by reference" übergeben.
- Die Datenübertragungsrichtung lässt sich mittels *intent*-Angabe steuern.

C

```
void sub1(...);
...
sub1(...);
...
void sub1(...) { ... }
datentyp fun1(...);
...
var = fun1(...);
...
datentyp fun1(...) { ... return
...; }
```

- Prinzipiell werden Parameter per "call by value" übergeben.
- "call by reference" wird mittels Übergabe von Zeigern realisiert.

68. Einfache Datentypen

Fortran95	**C**	**Bytes**
-----	unsigned-Datentypen	...
integer	int	4
integer(kind=1)	signed char	1
integer(kind=2)	short int	2
integer(kind=4)	int	4
integer(kind=8)	long int	8
real	float	4
double precision	double	8
real(kind=4)	float	4
real(kind=8)	double	8
complex	----- (ab ISO C99 -> *complex.h*-Bibliothek)	8
logical	----- (-> int-Zahlen, ab ISO C99 -> *stdbool.h*-Bibliothek)	4
character	unsigned char	1
character(len=...)	unsigned char[...]	...

(Zu beachten ist, dass die Byteanzahl der Datentypen teilweise maschinenabhängig ist. Die *kind*-Angaben sind zudem auch compilerabhängig.)

69. Variablendeklaration

Fortran95	**C**
datentyp :: var1, var2, ...	*datentyp* var1, var2, ...

70. Konstanten

Fortran95	**C**
'Hallo' oder "Hallo"	"Hallo"
'A', "A"	Character: 'A', Zeichenkette: "A"
Arithmetische Konstanten höherer Genauigkeit sind zwingend mit einer speziellen Datentypendung zu versehen	-----

71. Benannte Konstanten

Fortran95	**C**
datentyp, parameter :: konstante = *wert*	const *datentyp* konstante = *wert* oder #define konstante *wert*

72. Operatoren

72.1. Arithmetische Operatoren

Fortran95	C
+	+
-	-
*	*
/	/
**	----- (-> Bibliotheksfunktion)
---- (-> intrinsic functions)	% (Modulo)

72.2. Vergleichsoperatoren

Fortran95	C
==	==
<	<
<=	<=
>	>
>=	>=
/=	!=

72.3. Logische Operatoren

Fortran95	**C**
.AND.	&&
.OR.	||
.NOT.	!
.EQV.	-----
.NEQV.	-----

72.4. Stringverknüpfung

Fortran95	**C**
//	----- (-> Bibliotheksfunktion)

72.5. Bitoperatoren

Fortran95	**C**
----- (-> intrinsic functions)	<<, >>, &, |, ^, ~

72.6. Weitere Operatoren

Fortran95	**C**
-----	++, --, +=, -=, etc.

73. Zusammengesetzte Datentypen

Fortran95	**C**
type :: *name* sequence integer :: a ... end type *name* ... type(*name*) :: var var%a = *wert*	struct *name* { int a; ... }; ... struct *name* var; var.a = *wert*;

74. Felder

Fortran95	**C**
datentyp, dimension(*zahl*) :: var	*datentyp* var[*zahl*]
datentyp, dimension(*zahl1*, *zahl2*, ...) :: var;	*datentyp* var[*zahl1*][*zahl2*]...;
• Ein Feldindex startet standardmäßig bei 1. Dieser Indexstartwert kann aber vom Programmierer verschoben werden. • Ein mehrdimensionales Feld wird spaltenweise gespeichert. • Fortran95 kennt standardmäßig viele Möglichkeiten mit Feldern (Vektoren und Matrizen) zu hantieren (z.B. Feldkonstruktor, *reshape*, *where*, Zugriff auf eine Teilmenge eines Feldes, etc.).	• Ein Feldindex starten immer bei 0. • Ein mehrdimensionales Feld wird zeilenweise gespeichert.
var(*zahl*) = *wert*	var[*zahl*] = *wert*;

Fortran95	**C**
var(*zahl1*, *zahl2*) = *wert*	var[*zahl1*][*zahl2*] = *wert*;
var = *wert*	----- (-> mittels Schleifendurchlauf über das ganze Feld)
var(*zahl1*:*zahl2*) = *wert*	----- (-> mittels Schleifendurchlauf über die Feldteilmenge)

75. Zeichenketten

Fortran95

character(len=*zahl*) :: *var*

C

char *var*[*zahl*]

Zeichenketten werden (intern) mit einem \0-Zeichen terminiert. Dies muss bei der Variablendeklaration hinsichtlich der Feldlänge berücksichtigt werden.

76. Blöcke, Verzweigungen, Schleifen

76.1. Block

Fortran95	**C**
	{ ... }
...	

76.2. Verzweigungen

Fortran95	**C**
	if (*bedingung*) *anweisung*;
if (*bedingung*) *anweisung*	
if (*bedingung*) then ... end if	if (*bedingung*) { ... }
if (*bedingung*) then ... else ... end if	if (*bedingung*) { ... } else { ... }
-----	... ? ... : ...
select case (...) case (...) ... case (...) ... ... case default ... end select	switch (...) { case ...: ... break; case ...: ... break; ... default: ... }

76.3. Schleifen

Fortran95	**C**
do *schleifenvar* = *anfangswert, endwert, schrittweite* ... end do	for (*anfangsbedingung*; *endbedingung*; *durchgangsanweisung*) { ... }
do while (...) ... end do	while (...) { ... }
do ... if (...) exit end do	do { ... } while (...);

76.4. Sonstiges

Fortran95	**C**
stop	#include <stdlib.h> exit(*status*); oder im Hauptprogramm return;
exit	break;
cycle	continue;

77. Ein-, Ausgabe

Fortran95	**C**
	#include <stdio.h> printf(...);
write(*, *format*) ...	
read(*, *format*) ...	#include <stdio.h> scanf(...);

78. Dateien

Fortran95

open (unit=*nr, ...)* write (*nr*, *format*) ... read (*nr*, *format*) ... ... close(unit=*nr*)

C

#include <stdio.h> ... FILE *f = fopen(...); fprintf (f, ...); fscanf(f, ...); ... fclose(f);

79. Zeiger

Fortran95

datentyp, pointer :: var
... *datentyp*, pointer :: var1
=> null(); *datentyp*, target ::
var2 = *wert*; ... var1 => var2
write(*,*) var1

C

datentyp *var;

... *datentyp* *var1 = 0; *da-
tentyp* var2 = *wert*; ... var1 =
&var2; printf("%d", *var1);

80. Anwendungsbeispiele

80.1. Dreiecksberechnung

81. Aufgabe

Es sollen einige charakteristische Dreieckswerte berechnet werden. Der Programmanwender gibt die Koordinatenwerte (x, y) für die Dreieckseckpunkte P_1, P_2 und P_3 vor.

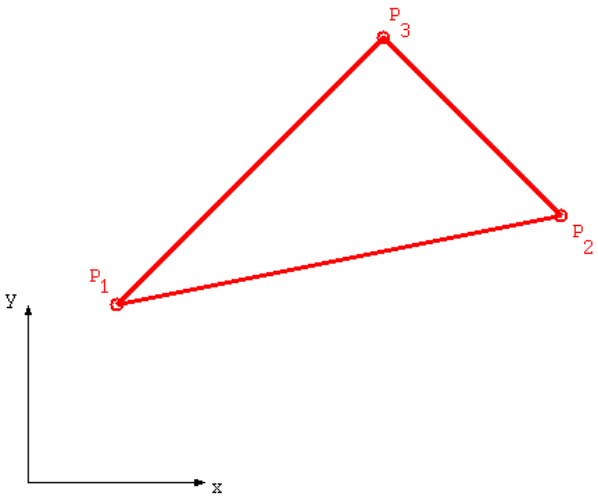


Abb. 57

Das Programm berechnet u.a. folgende Werte und übermittelt diese an die Standardausgabe:

- Längen der Dreiecksschenkel und Dreiecksumfang
- Innenwinkel

- Fläche
- Umkreis (Mittelpunkt und Radius)
- Inkreis (Mittelpunkt und Radius)
- Schwerpunkt

82. Grundlagen

Die Dreiecksberechnung erfolgt in diesem Anwendungsbeispiel hauptsächlich mittels Vektorrechnung.

Näheres zu Dreiecken und zur Vektorrechnung ist folgenden Enzyklopädieartikeln und Büchern zu entnehmen:

- WIKIPEDIA: DREIECK[1]
- WIKIPEDIA: VEKTOR[2]
- FORMELSAMMLUNG MATHEMATIK: GEOMETRIE[3]
- FORMELSAMMLUNG MATHEMATIK: TRIGONOMETRIE[4]
- MATHEMATIK: SCHULMATHEMATIK[5]
- MATHEMATIK: LINEARE ALGEBRA[6]
- ING MATHEMATIK: VEKTOREN[7]

---

1 HTTP://DE.WIKIPEDIA.ORG/WIKI/DE%3ADREIECK%20
2 HTTP://DE.WIKIPEDIA.ORG/WIKI/DE%3AVEKTOR%20
3 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORMELSAMMLUNG%20MATHEMATIK%3A%20GEOMETRIE
4 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORMELSAMMLUNG%20MATHEMATIK%3A%20TRIGONOMETRIE
5 HTTP://DE.WIKIBOOKS.ORG/WIKI/MATHEMATIK%3A%20SCHULMATHEMATIK
6 HTTP://DE.WIKIBOOKS.ORG/WIKI/MATHEMATIK%3A%20LINEARE%20ALGEBRA
7 HTTP://DE.WIKIBOOKS.ORG/WIKI/ING%20MATHEMATIK%3A%20VEKTOREN

82.1. Koordinatenwerte ---> Richtungsvektoren

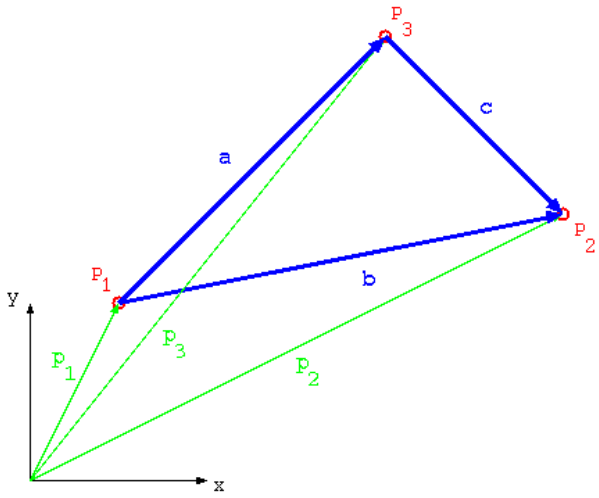


Abb. 58

$\mathbf{a} = \mathbf{p}_3 - \mathbf{p}_1$

$\mathbf{b} = \mathbf{p}_2 - \mathbf{p}_1$

$\mathbf{c} = \mathbf{b} - \mathbf{a}$

82.2. Seitenlängen und Umfang

$a_{norm} = |\mathbf{a}|$

$b_{norm} = |\mathbf{b}|$

$c_{norm} = |\mathbf{c}|$

$U = a_{norm} + b_{norm} + c_{norm}$

Bedingung: $a_{norm} \neq 0$, $b_{norm} \neq 0$, $c_{norm} \neq 0$

82.3. Winkel

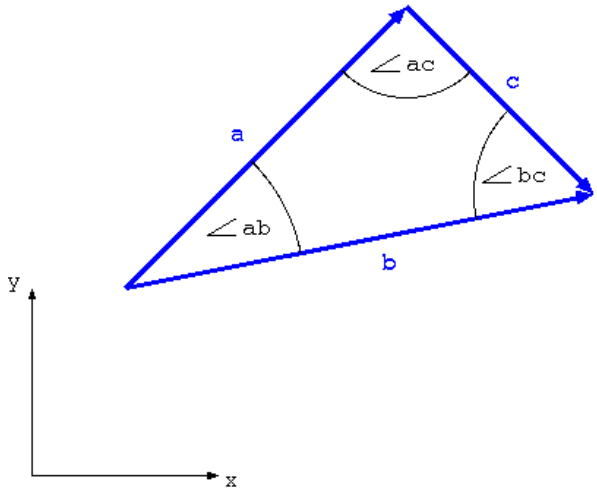


Abb. 59

$\angle ab = \arccos \frac{\mathbf{a} \cdot \mathbf{b}}{a_{norm} b_{norm}}$

$\angle bc = \arccos \frac{\mathbf{b} \cdot \mathbf{c}}{b_{norm} c_{norm}}$

$\angle ac = \pi - \angle ab - \angle bc$

Bedingung: $\angle ab \neq 0$, $\angle ac \neq 0$, $\angle bc \neq 0$

82.4. Fläche

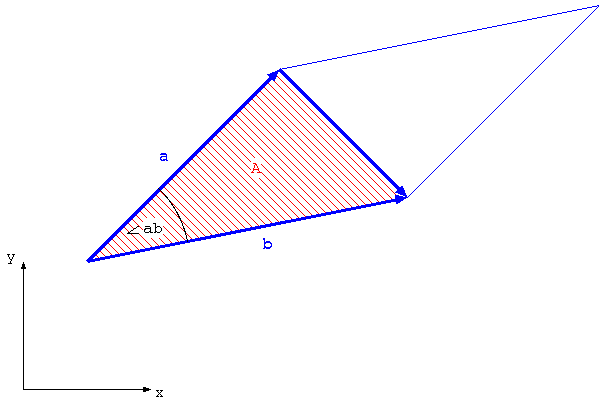


Abb. 60

Es gilt

$A_{Parallelogramm} = |\mathbf{a} \times \mathbf{b}| = a_{norm} b_{norm} \sin \angle ab$

und somit

$A = \frac{A_{parallelogramm}}{2} = \frac{a_{norm} b_{norm} \sin \angle ab}{2}$

82.5. Umkreis

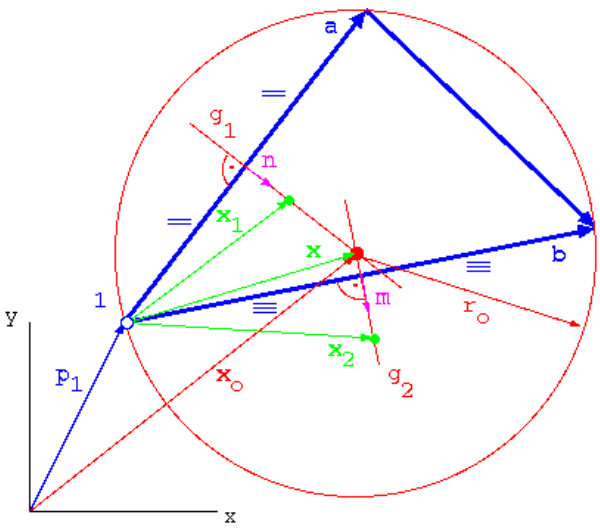


Abb. 61

Normalen:

$\mathbf{a} \cdot \mathbf{n} = 0$

$\mathbf{b} \cdot \mathbf{m} = 0$

$|\mathbf{n}| = 1$

$|\mathbf{m}| = 1$

$\Rightarrow$

$$n_1 = -a_2\sqrt{\frac{1}{a_1^2+a_2^2}}$$

$$n_2 = a_1\sqrt{\frac{1}{a_1^2+a_2^2}}$$

$$m_1 = -b_2\sqrt{\frac{1}{b_1^2+b_2^2}}$$

$$m_2 = b_1\sqrt{\frac{1}{b_1^2+b_2^2}}$$

Geradenschnittpunkt:

$$g1: \mathbf{x}_1 = \frac{\mathbf{a}}{2} + t_1\mathbf{n}$$

$$g2: \mathbf{x}_2 = \frac{\mathbf{b}}{2} + t_2\mathbf{m}$$

Der Umkreismittelpunkt ergibt sich als Schnittpunkt dieser beiden Geraden: $\mathbf{x} = \mathbf{x}_1 = \mathbf{x}_2$

$$\Rightarrow$$

$$t_2 = \frac{n_1(a_2-b_2)+n_2(b_1-a_1)}{2(m_2n_1-m_1n_2)}$$

Bedingung: $m_2n_1 - m_1n_2 \neq 0$

Umkreismittelpunkt und -radius:

$$\mathbf{x}_o = \mathbf{p}_1 + \frac{\mathbf{b}}{2} + t_2\mathbf{m}$$

$$r_o = \left|\frac{\mathbf{b}}{2} + t_2\mathbf{m}\right|$$

82.6. Inkreis

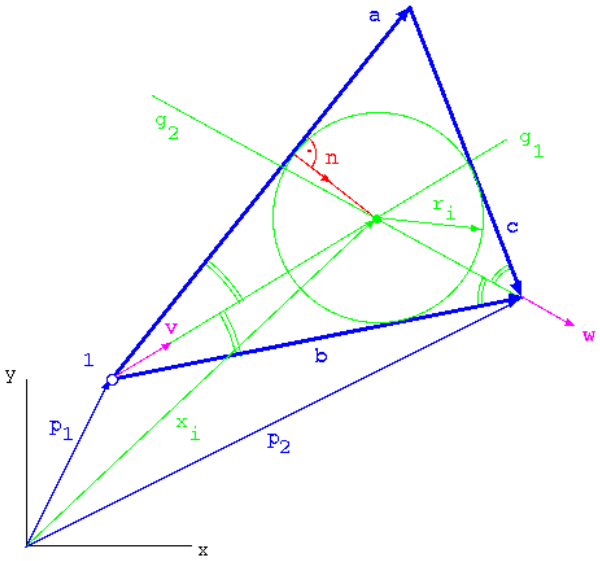


Abb. 62

$$\mathbf{v} = \frac{1}{2}\left(\frac{\mathbf{a}}{a_{norm}} + \frac{\mathbf{b}}{b_{norm}}\right)$$

$$\mathbf{w} = \frac{1}{2}\left(\frac{\mathbf{b}}{b_{norm}} + \frac{\mathbf{c}}{c_{norm}}\right)$$

$g1: \mathbf{x}_1 = t_1\mathbf{v}$

$g2: \mathbf{x}_2 = \mathbf{b} + t_2\mathbf{v}$

Der Inkreismittelpunkt ergibt sich als Schnittpunkt dieser beiden Geraden: $\mathbf{x} = \mathbf{x}_1 = \mathbf{x}_2$

$\Rightarrow$

$t_2 = \frac{b_1 v_2 - b_2 v_1}{w_2 v_1 - w_1 v_2}$

Bedingung: $w_2 v_1 - w_1 v_2 \neq 0$

Inkreismittelpunkt:

$\mathbf{x}_i = \mathbf{p}_2 + t_2 \mathbf{w}$

Inkreisradius:

$r_i = \frac{2A}{a_{norm} + b_{norm} + c_{norm}}$

82.7. Schwerpunkt

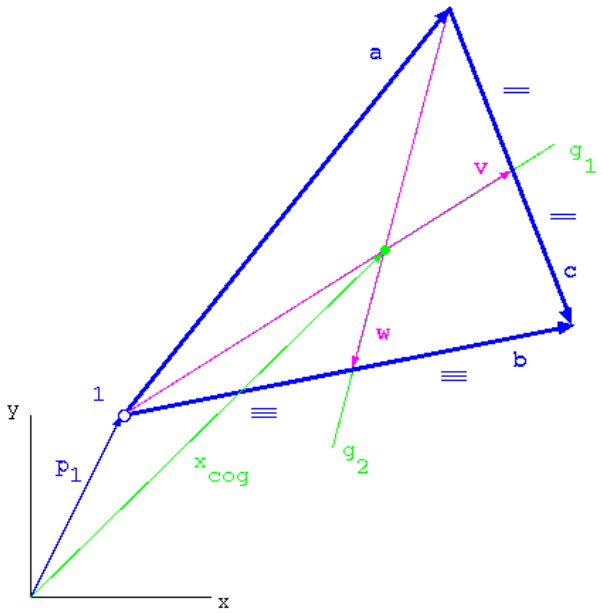


Abb. 63

$\mathbf{v} = \mathbf{a} + \frac{\mathbf{c}}{2}$

$\mathbf{w} = \frac{\mathbf{b}}{2} - \mathbf{a}$

$g1 : \mathbf{x}_1 = t_1\mathbf{v}$

$g2 : \mathbf{x}_2 = \mathbf{a} + t_2\mathbf{w}$

Der Dreiecksschwerpunkt ergibt sich als Schnittpunkt dieser beiden Geraden: $\mathbf{x} = \mathbf{x}_1 = \mathbf{x}_2$

$t_2 = \frac{a_1 v_2 - a_2 v_1}{w_2 v_1 - w_1 v_2}$

Bedingung: $w_2 v_1 - w_1 v_2 \neq 0$

$\Rightarrow$

$\mathbf{x} = \mathbf{a} + t_2\mathbf{w}$

$\mathbf{x}_{cog} = \mathbf{p}_1 + \mathbf{x}$

83. Code

- MAKEFILE[1]
- TRIANGLE.F90[2]
- MODUL1.F90[3]

1 http://de.wikibooks.org/wiki/Fortran%3A_Anhang_B%3A_Dreiecksberechnung%3A_Makefile%20%20

2 http://de.wikibooks.org/wiki/Fortran%3A_Anhang_B%3A_Dreiecksberechnung%3A_Triangle.f90%20

3 http://de.wikibooks.org/wiki/Fortran%3A_Anhang_B%3A_Dreiecksberechnung%3A_Modul1.f90%20

84. Screenshots

```
********** DREIECKSBERECHNUNG **********

 Eingabe:
  Punkt 1 (x y): 10.5 20.56
  Punkt 2 (x y): 100.6 70.5
  Punkt 3 (x y): 110 15.5

*** Ergebnisse ***
  Eingabe:
   P1                  =    10.50000       20.56000
   P2                  =    100.6000       70.50000
   P3                  =    110.0000       15.50000
  Berechnet:
   a                   =    99.50000      -5.059999
   b                   =    90.10000       49.94000
   c                   =   -9.400002       55.00000
   Länge a             =    99.62858
   Länge b             =    103.0146
   Länge c             =    55.79749
   Umfang              =    258.4407
   Winkel ab           =   0.5569285     rad
   Winkel ab           =    31.90965     °
   Winkel ac           =    1.350712     rad
   Winkel ac           =    77.39011     °
   Winkel bc           =    1.233952     rad
   Winkel bc           =    70.70023     °
   Fläche              =    2712.468     ^2
   Umkreismittelpunkt  =    61.13598       35.45197
   Umkreisradius       =    52.78043
   Inkreismittelpunkt  =    84.89420       37.79489
   Inkreisradius       =    20.99103
   Schwerpunkt         =    73.70000       35.52000
```

Abb. 64

84.1. Kettenlinie

85. Aufgabe

Der Programmbenutzer gibt die Kettenlänge L, die Abstände der beiden Abhängepunkte in x- und y-Richtung (b und h), sowie das spezifische Kettengewicht q vor. Aus diesen Angaben werden diverse Daten (Seilparameter, Kräfte, Durchhang, ...) für die Kette berechnet. Zusätzlich wird die aus den Angaben resultierende Kettenlinie grafisch am Bildschirm ausgegeben. Diese grafische Ausgabe erfolgt mit Hilfe der DISLIN-Grafikbibliothek.

86. Grundlagen

- SEILSTATIK[1]
- siehe NULLSTELLENBESTIMMUNG NACH NEWTON-RAPHSON[2]

1 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20ANHANG%20B%3A%20KETTENLINIE%3A%20SEILSTATIK%20

2 HTTP://DE.WIKIPEDIA.ORG/WIKI/DE%3ANEWTON-{}VERFAHREN%20

87. Code

- MAKEFILE[1]
- CATENARY.F90[2]
- CATMOD1.F90[3]
- CATDISL.F90[4]
- dislin.f90 (diese Datei ist Bestandteil der DISLIN-Bibliothek)

1 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_ANHANG_B%3A_KETTENLINIE%3A_MAKEFILE%20%20

2 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_ANHANG_B%3A_KETTENLINIE%3A_CATENARY.F90%20

3 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_ANHANG_B%3A_KETTENLINIE%3A_CATMOD1.F90%20

4 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_ANHANG_B%3A_KETTENLINIE%3A_CATDISL.F90%20

88. Screenshots

```
********** KETTENLINIE **********

 Eingabe:
  Seillänge                  L: 250
  Breite                     b: 200
  Höhe                       h: 30.5
  Spezifisches Kettengewicht q: 90.8

*** Ergebnisse ***
  Eingabe:
   L                              =    250.0000
   b                              =    200.0000
   h                              =    30.50000
   q                              =    90.80000
  Berechnet:
   Seilparameter a (1.Näherung)   =    85.98637
   Seilparameter a                =    86.07004
   x1                             =   -89.44689
   x2                             =    110.5531
   y1                             =    136.8847
   y2                             =    167.3847
   s1                             =   -106.4396
   s2                             =    143.5604
   phi1                           =  -0.8908163     rad
   phi1                           =   -51.04002     °
   phi2                           =    1.030716     rad
   phi2                           =    59.05568     °
   F1                             =    12429.13
   F2                             =    15198.54
   H=H1=H2                        =    7815.161
   V1                             =   -9664.712
   V2                             =    13035.29
   Durchhang f                    =    65.45425

 <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
 <<                  END OF DISLIN / VERSION  9.0                              <<
 <<  Date    : 17.02.2006  Time    : 20:14:25  Pageformat: DA4L  Fileformat:  PNG  <<
 <<  Vectors : 1037        Warnings: 0                                           <<
 <<  Metafile: dislin.png                                                        <<
 <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

Abb. 65

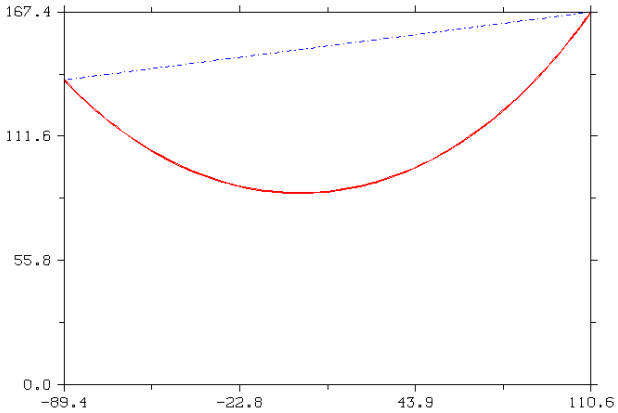


Abb. 66

88.1. Inverse Matrix

89. Aufgabe

Der Programmbenutzer gibt eine reelle *nxn*-Matrix vor. Das Programm berechnet dazu die inverse Matrix.

90. Grundlagen

Mit Hilfe der LAPACK-Bibliothek ist die Berechnung einer inversen Matrix sehr einfach durchzuführen. Aus diesem Grund soll bei diesem Beispiel das Hauptaugenmerk auf das Einbinden einer C-Bibliothek in ein Fortran-Programm gelegt werden. Für diesen Zweck bietet sich hier die ncurses-Bibliothek an. Ncurses ist eine Bibliothek zur Erstellung von TUIs (Text User Interfaces). In diesem Beispiel soll die Datenein- und -ausgabe mit Hilfe der ncurses-Bibliothek über einfache Formulare menügeführt erfolgen.

- Berechnung der inversen Matrix:
  - INVERSE MATRIX[1]
  - LU-ZERLEGUNG[2]
  - LAPACK[3]
- Gestaltung der Benutzerschnittstelle: NCURSES[4]

1 HTTP://DE.WIKIPEDIA.ORG/WIKI/INVERSE%20MATRIX%20
2 HTTP://DE.WIKIPEDIA.ORG/WIKI/LR-{}ZERLEGUNG%20
3 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A%20LAPACK%20
4 HTTP://DE.WIKIBOOKS.ORG/WIKI/NCURSES%20

91. Code

92. Screenshots

93. Debugger

Werden Programme länger und komplexer, so kann der Einsatz eines Debuggers das Auffinden von logischen Fehlern erleichtern.

94. Der GNU Debugger

Der *gdb* (GNU Debugger) wird besonders im Open Source-Bereich im Zusammenspiel mit der GCC verwendet. Die Anwendung im Zusammenhang mit *gfortran*-compilierten Programmen ist relativ einfach, wenn auch nicht so problemlos wie bei C- oder C++-Programmen.

94.1. gdb und gfortran

Das Programm muss mit dem Optionsschalter `-g` erstellt werden. *gfortran* kennt noch andere erweiterte Optionen, die spezialisiertere Ergebnisse liefern. Für einfache Beispiele wie sie in diesem Buch vorzufinden sind reicht aber die `-g`-Option vollkommen aus.

gfortran -g -o bsp bsp.f90

Aufruf des Debuggers mittels

gdb bsp

Zu beachten ist, dass nach dem Start des Debuggers ein Breakpoint

b MAIN__

gesetzt wird, ansonsten findet der Debugger das zu debuggende Fortran-Programm nicht. Danach wird der Debuggerlauf mit

run

gestartet.

Einige wichtige *gdb*-Befehle:

`b <zahl>`	Breakpoint in der Zeile `<zahl>` setzen
`list`	Codelisting
`next`	Zur nächsten Zeile springen
`step`	Sprung in eine Subroutine
`print <var>`	Gibt den Wert der Variablen `<var>` aus
`c`	Das Programm fortsetzen (continue)
`until <zahl>`	Programm bis zur Zeile `<zahl>` ausführen
`quit`	Debugger beenden

94.2. gdb und g95

Das Debuggen von *g95*-compilierten Programmen funktioniert beinahe identisch zur Vorgehensweise mit *gfortran*. Allerdings ist zu beachten, dass ein Breakpoint

*b MAIN_*

(nur mit einem Unterstrich) vor dem `run`-Befehl gesetzt wird.

95. Der Intel Debugger

Die Firma Intel liefert bei ihren Fortran- und C++-Compilern den Kommandozeilendebugger *idb* mit.

Gestartet wird der Intel-Debugger mittels

idb -gdb bsp

Der Debugger befindet sich dann im *gdb*-Modus. Wird der *idb* ohne `-gdb`-Schalter aufgerufen, so wird er im *dbx*-Modus gestartet. Prinzipiell ist es ziemlich egal, in welchem Modus debugged wird. Die Befehlsbezeichungen und der Befehlsumfang unterscheiden sich etwas. Vor dem `run` sollte ein Breakpoint gesetzt werden. Das allerdings ganz normal mit Angabe einer Zeilennummer.

96. Quellcodedokumentation

96.1. ROBODoc

ROBODoc ist freie Software unter der GNU General Public License. Dieses Tool verwendet die normalen Kommentarzeichen der jeweiligen Programmiersprache und nutzt für die Dokumentationsgenerierung eigene Auszeichnungstags innerhalb dieser Kommentare.

96.2. Ein einfaches Beispiel

96.2.1. Beispielcode

Fortran 90/95-Code (free source form)

```
!****h* Beispielprogramm/circle
! NAME
!   circle
!
! DESCRIPTION
!   Modul fuer Kreisfunktionen
!
! AUTHOR
!   Intruder
!
! CREATION DATE
!   04.08.2007
!******
module circle
  implicit none
```

```
  !****d* circle/pi
  ! NAME
  !  pi
  !
  ! DESCRIPTION
  !  pi = 3.14
  !******
  real, parameter :: pi = 3.14
  contains
    !****f* circle/area
    ! NAME
    !   area
    !
    ! DESCRIPTION
    !   Berechnet Kreisflaeche
    !
    ! INPUTS
    !   r ... Radius (real)
    !
    ! RESULT
    !    Kreisflaeche (real)
    !******
    real function area( r )
      implicit none
      real, intent( in ) :: r

      area = r ** 2 * pi
    end function area
end module circle
```

Fortran 90/95-Code (free source form)

```
!****h* Beispielprogramm/main
! NAME
!   main
!
! DESCRIPTION
!   Hauptprogramm
!
! AUTHOR
!   Intruder
!
! CREATION DATE
!   04.08.2007
!******
program main
  use circle
```

```
  implicit none

  real            :: r, a

  read( *, * ) r
  a = area( r )
  write( *, * ) "Flaeche = ", a
end program main
```

96.2.2. Erstellen der Dokumentation

Erstellen der Dokumentation im HTML-Format als Multidokument mit Index:

```
robodoc --src . --doc doc --multidoc --html --index
```

96.2.3. Screenshot

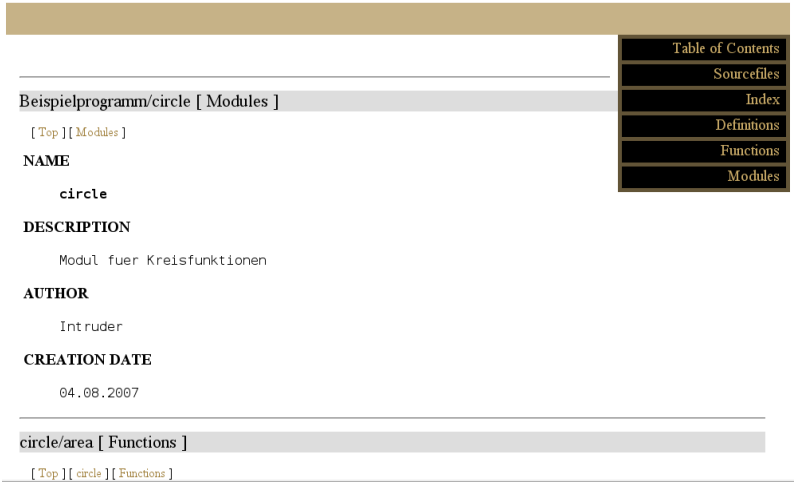


Abb. 67

96.2.4. Kurze Erläuterung

ROBODoc filtert die erforderlichen Angaben für die Dokumentation aus den Fortran-Kommentaren heraus. Zu diesem Zweck sind in den Kommentaren *Header* einzubauen. Diese *Header* bestehen aus

- begin marker
- items
- end marker

Ein *begin marker* beginnt immer mit 4 Sternchen, dann kommt ein Buchstabe als Elementkennzeichner. Es folgt ein Stern und dann

Angaben zur Stellung des Elementes in der Dokumentationshierarchie.

In diesem Beispiel wurden folgende Elementkennzeichner verwendet:

- `h` ... Modul
- `f` ... Funktion
- `d` ... Konstante (Definition)

Der Hierarchiebaum wird durch die strikte Angabe von

```
übergeordnetes Element/aktuelles Element
```

erstellt.

Die verschiedenen Items sind dann unterhalb dieses *begin markers* durch Schlüsselworte gekennzeichnet:

- `NAME`
- `DESCRIPTION`
- `AUTHOR`
- etc.

Sie dienen zur konkreten Beschreibung des jeweiligen Elementes.

Abgeschlossen wir ein solcher Dokumentations-Header durch den *end marker*. Dieser wird durch mindestens drei Sternchen gebildet.

Zwecks Erstellung der Dokumentation sind viele Optionen verfügbar. Die unbedingt erforderlichen Angaben sind

- `--src` mit Angabe der Quelldatei bzw. dem Verzeichnis, in dem die Quelldateien liegen

- `--doc` mit einem Vornamen für die Dokumentationsdateien bzw. der Bezeichnung des gewünschten Dokumentationsverzeichnisses
- eine Angabe zu der Dokumentationsform:
  - `--multidoc` ... Die Dokumentation wird in Form mehrerer Einzeldateien in des Dokumentationsverzeichnis geschrieben
  - `--singledoc` ... Der Dokumentationsinhalt wird in eine einzige Datei geschrieben
- die Angabe des Dokumentationsformates:
  - `--html` ... HTML
  - `--rtf` ... RTF
  - `--latex` ... LaTeX
  - `--dbxml` ... XML DocBook

Für dieses Beispielprojekt wird auch noch ein Index erstellt (`--index`).

Detaillierte Informationen zu ROBODoc sind unter dem nachfolgend angeführten Weblink zur ROBODoc-Homepage abrufbar.

96.3. Weblinks

- ROBODoc[1]

96.4. Natural Docs

Auch der Dokumentationsgenerator *Natural Docs* unterstützt bereits Fortran ab dem Standard 90/95 (free source form) in einer Basisvariante. *Natural Docs* ist in der Programmiersprache Perl verfasst. Als Softwarelizenz wurde die *General Public License*

1 http://www.xs4all.nl/%7Erfsber/Robo/robodoc.html

gewählt. Zum Zeitpunkt der Kapitelerstellung war die Version 1.35 aktuell.

96.5. Ein einfaches Beispiel

Fortran 90/95-Code (free source form)

```
! Section: Beispielprogramm
!   Autor: Intruder, Datum: 04.08.2007
! Group: circle
!   Modul fuer Kreisfunktionen
module circle
  implicit none
  ! Constant: pi
  !   pi = 3.14
  real, parameter :: pi = 3.14
  contains
    ! Function: area
    !   Berechnet Kreisflaeche
    !
    ! Parameters:
    !   r ... Radius (real)
    !
    ! Returns:
    !    Kreisflaeche (real)
    real function area( r )
      implicit none
      real, intent( in ) :: r

      area = r ** 2 * pi
    end function area
end module circle
! Group: main
!   Hauptprogramm
program main
  use circle
  implicit none

  real            :: r, a

  read( *, * ) r
  a = area( r )
  write( *, * ) "Flaeche = ", a
```

```
end program main
```

Erstellung der HTML-Dokumentation:

```
NaturalDocs -i . -o HTML bsp -p .
```

Screenshot:

Abb. 68

96.6. Kurze Erläuterung

Natural Docs filtert die benötigten Informationen mit Hilfe von Schlüsselwörtern aus den Kommentarbereichen der Fortrandateien. Die im Beispielprogramm verwendeten Schlüsselwörter sind:

- Section
- Group
- Constant
- Function
- Parameters
- Returns

Groß- Kleinschreibung spielt bei diesen Schlüsselwörtern kein Rolle. Abgeschlossen wird ein Schlüsselwort mit einem Doppelpunkt.

Beim Ausgabeschalter (-o) muss neben einem Ausgabeverzeichnis auch noch ein Ausgabeformat angegeben werden. Möglich sind

- HTML
- FramedHTML

Beim Schalter -p ist ein (Projekt)Verzeichnis anzugeben, wo *Natural Docs* einige benötigte Dateien ablegen kann, die nicht direkt zur erstellten Dokumentation gehören.

Für weitergehende Infos zu *Natural Docs* wird auf die unten angegebene Webpräsenz zu dieser Software verwiesen.

96.7. Weblinks

- NATURAL DOCS[2]

96.8. Doxygen

Mit Version 1.5.4 vom 27. Oktober 2007 kann auch *doxygen* aus Fortran 90/95-Dateien Dokumentationen erstellen.

2 HTTP://WWW.NATURALDOCS.ORG/

96.9. Ein einfaches Beispiel

Fortran 90/95-Code (free source form)

```
!> \file testprogramm.f90
!! \brief Datei fuer geometrische Berechnungen
!> Modul: circle
!> \author Intruder
!> \date 28.10.2007
module circle
  implicit none
  real, parameter :: pi = 3.14  !< PI
  contains
    !> Berechnet Kreisflaeche
    !> \param r ... Radius
    !> \return Kreisflaeche
    real function area( r )
      implicit none
      real, intent( in ) :: r

      area = r ** 2 * pi
    end function area
end module circle
!> Hauptprogramm
program main
  use circle
  implicit none

  real            :: r, a

  read( *, * ) r
  a = area( r )
  write( *, * ) "Flaeche = ", a
end program main
```

Erstellen einer Konfigurationsdatei:

```
doxygen -g Doxyfile
```

Anschließend sollte man die Doxyfile bearbeiten und nach seinen eigenen Wünschen anpassen (Projektname, Outputordner etc.).

Wichtig ist, dass das Flag OPTIMIZE_FOR_FORTRAN auf YES gesetzt wird.

Erstellen der Dokumentation (standardmäßig im HTML- und LaTeX-Format, optional auch XML, RTF und man-pages):

```
doxygen Doxyfile
```

Screenshot (HTML-Variante):


real area (r)
Berechnet Kreisflaeche.

Variables

real, parameter pi = 3.14
PI.

Detailed Description

Modul: circle.

Author:
Intruder

Date:
28.10.2007

Function Documentation

real circle::area (real,intent(in) r)

Berechnet Kreisflaeche.

Parameters:
r ... Radius

Returns:
Kreisflaeche

Abb. 69

96.10. Weblinks

- DOXYGEN-HOMEPAGE[3]

3 HTTP://WWW.STACK.NL/~{}DIMITRI/DOXYGEN

- Ankündigung der Fortran 90-Unterstützung in *doxygen* 1.5.4[4]
- DOUG - Doxygen F90[5]

4 http://www.stack.nl/~{}dimitri/doxygen/changelog.html

5 http://www.dougdevel.org/index.php?page=doxygen

97. Make & Co.

Abb. 70

97.1. Einleitung

Hier wird keine Einführung in die Verwendung und Syntax von make- und config-Tools geboten, sondern nur kurz auf einige Spezialitäten hingewiesen, die bei den ersten Einsatzversuchen derartiger Werkzeuge im Zusammenhang mit Fortran zu beachten sind.

Grundsätzlich kann bei der Erstellung von Makefiles und Konsorten eine ähnliche Vorgehensweise wie bei konventionellen C-Programmen gewählt werden. Es ist bei Fortran-Programmen jedoch zu bedenken, dass bei Verwendung von Modulen `mod`-Dateien generiert werden (ab Fortran 90). Diese `mod`-Dateien sind in weiterer Folge für das Kompilieren von moduleinbindenden Quellcodedateien und den Linkvorgang von entscheidender Bedeutung. Somit ist bei hierarchisch verzweigten Quellcodeverzeichnisbäumen Obacht zu geben, dass jeweils auch Zugriff zu diesen `mod`-Dateien gegeben ist. Dies kann geschehen durch

- geeigneten Aufbau der Makefiles,
- durch Verwendung von Tools, die solche Abhängkeiten automatisch auflösen
- oder auch durch explizite Bekanntgabe der entsprechenden Pfade an Compiler und Linker.

97.2. Explizite Bekanntgabe von Modulpfaden

97.2.1. gfortran

Standardmäßig werden include- und mod-Dateien im aktuellen Verzeichnis gesucht. Die Suche kann aber mit folgendem Compilerschalter auf andere Pfade ausgedehnt werden:

- `-I...`: Suchpfad für
  - include-Dateien
  - mod-Dateien

    erweitern.

Standardmäßig werden mod-Dateien in das aktuelle Verzeichnis geschrieben. Dieses Verhalten kann mit folgendem Schalter geändert werden:

- `-J...`: Legt Verzeichnis fest, in das die mod-Dateien geschrieben werden, gleichzeitig auch Suchpfad für mod-Dateien.

  (Alias für `-M...` um Konflikte mit bisherigen GCC-Optionen zu vermeiden)

97.2.2. g95

Standardmäßig werden include- und mod-Dateien im aktuellen Verzeichnis gesucht. Die Suche kann aber mit folgendem Compilerschalter auf andere Pfade ausgedehnt werden:

- `-I...`: Suchpfad für
  - include-Dateien
  - mod-Dateien

    erweitern.

Auch der g95-Compiler kennt einen `-M`-Schalter. Dieser hat aber eine komplett andere Bedeutung als beim gfortran-Compiler. Stattdessen gibt es beim g95-Compiler den Schalter:

- `-fmod=...`: Legt Verzeichnis fest, in das die mod-Dateien geschrieben werden, gleichzeitig auch Suchpfad für mod-Dateien.

97.2.3. ifort

- `-I...`: Suchpfad für
  - include-Dateien
  - mod-Dateien

    erweitern.
- `-module ..`: Legt Verzeichnis fest, in das die mod-Dateien geschrieben werden, gleichzeitig auch Suchpfad für mod-Dateien.

97.2.4. Sun f90/f95

- `-M...`: Fügt den angegebenen Pfad zum Modulsuchpfad hinzu.
- `-moddir=...`: mod-Dateien werden in das angegebene Verzeichnis geschrieben, gleichzeitig auch Suchpfad für mod-Dateien.

97.3. GNU Make

GNU Make erkennt derzeit leider nur FORTRAN 77-Dateien mit der Endung .f automatisch. Für "free source form"-Fortran-Programme sind daher einige vorbereitende Arbeiten nötig, um dann auch die Vorteile (und Nachteile) der impliziten Anwendung von "Pattern Rules" genießen zu dürfen. Werden alle Makeschritte

für Fortran-Dateien explit vorgegeben, dann kann man sich dies natürlich sparen.

97.3.1. Ein einfaches Beispiel

Es sei ein einfaches Beispiel gegeben, das eine FORTRAN 77-, eine Fortran 2003- und eine C-Datei enthält. Diese Dateien liegen im selben Verzeichnis.

Quellcode-Dateien

main.f03: **Fortran 2003-Code**

```
! Das Hauptprogramm
program main
  implicit none
  interface
    function addition( a, b ) bind( c )
      use, intrinsic :: iso_c_binding
      real( kind = c_float ), value :: a
      real( kind = c_float ), value :: b
      real( kind = c_float )        :: addition
    end function addition

    subroutine sub()
    end subroutine sub
  end interface
  call sub()
  write (*,*) addition( 2.5, 3.3 )

! Ausgabe:
!    Summe =
!      5.8
end program main
```

func.c: **Programmcode**

```
/* Addiere zwei Zahlen */
float addition(float a, float b)
{
  return (a + b);
```

```
}
```

sub.f:

```
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

C  Eine einfache FORTRAN 77-Subroutine
      SUBROUTINE SUB
      WRITE( *, * ) 'Summe ='
      END


12345678901234567890123456789012345678901234567890123456789012345678901234567890
0   . |  1    .    2    .    3    .    4    .    5    .    6    .    7 |  .    8
```

Explizite Vorgabe der Makeschritte

Makefile:

```
FC = gfortran        # oder g95, ifort, ...
prog: main.o func.o sub.o
    $(FC) -o $@ $^
main.o: main.f03
    $(FC) -c $^
func.o: func.c
    $(CC) -c $^

sub.o: sub.f
    $(FC) -c $^
```

Nutzung von "Pattern Rules"

Makefile:

```
FC  = gfortran      # oder g95, ifort, ...
```

```
%.o: %.f03
    $(FC) -c $<
prog: main.o func.o sub.o
    $(FC) -o $@ $^
```

Die Generierung der Objektdateien aus den Quellcodedateien geschieht hier implizit. Für C- und FORTRAN 77-Dateien sucht sich GNU Make die entsprechenden Regeln aus seiner internen Regel-Datenbank. Für .f03-Dateien wurde der entsprechende Befehl hier von uns explizit durch eine "Pattern Rule" vorgegeben.

Die make-Ausgabe sieht so aus:

```
gfortran -c main.f03
cc    -c -o func.o func.c
gfortran   -c -o sub.o sub.f
gfortran -o prog main.o func.o sub.o
```

Solange die Quelldateien im selben Verzeichnis liegen, ist die Erstellung eines Makefiles ziemlich einfach. Wenn aber die Quelldateien gestaffelt in Unterverzeichnissen gespeichert sind und womöglich noch Abhängigkeiten von einem Unterverzeichis zum anderen gegeben sind, dann kann die ganze Sache ziemlich kompliziert werden. Im Anschluss wird eine einfache nichtrekursive Make-Variante gezeigt.

97.3.2. Nichtrekursive Make-Variante

Vor der Programmerstellung:

```
-- (D) projektverzeichnis
 |-- (F) Makefile
 |-- (F) module.mk
```

```
 |-- (F) main.f90
 |
 |-- (D) kreis
 | |-- (F) module.mk
 | |-- (F) kreis.f90
 | |-- (F) kreissegment.f90
 |
 |-- (D) quadrat
 | |-- (F) module.mk
 | |-- (F) quadrat.f90

(D) ... directory
(F) ... file
```

Nach der Programmerstellung durch Aufruf von `make`:

```
-- (D) projektverzeichnis
 |-- (F) Makefile
 |-- (F) module.mk
 |-- (F) main.f90
 |-- (F) main.o
 |-- (F) prog
 |-- (F) kreis.mod
 |-- (F) kreissegment.mod
 |-- (F) quadrat.mod
 |
 |-- (D) kreis
 | |-- (F) module.mk
 | |-- (F) kreis.f90
 | |-- (F) kreissegment.f90
 | |-- (F) kreis.o
 | |-- (F) kreissegment.o
 |
 |-- (D) quadrat
 | |-- (F) module.mk
 | |-- (F) quadrat.f90
 | |-- (F) quadrat.o
```

Quellcode-Dateien

main.f90: **Fortran 90/95-Code (free source form)**

```
program main
  use kreis
  use kreissegment
  use quadrat
  implicit none

  call k()
  call q()
  call ks()
end program main
```

kreis/kreis.f90: **Fortran 90/95-Code (free source form)**

```
module kreis
  implicit none

  private
  public :: k

  contains
    subroutine k()
      print *, "Ich bin ein Kreis!"
    end subroutine k
end module kreis
```

kreis/kreissegment.f90: **Fortran 90/95-Code (free source form)**

```
module kreissegment
  use kreis
  implicit none

  private
  public :: ks

  contains
    subroutine ks()
      call k()
      print *, "Hihi, war nur ein Scherz. Ich bin ein Kreissegment!"
    end subroutine ks
end module kreissegment
```

quadrat/quadrat.f90: **Fortran 90/95-Code (free source form)**

```
module quadrat
  implicit none
  private
  public :: q

  contains
    subroutine q()
      print *, "Ich bin ein Quadrat!"
    end subroutine q
end module quadrat
```

Makefile, Include-Dateien

Makefile:

```
FC       := g95

SRC      :=

OBJ      = $(subst .f90,.o,$(SRC))

%.o: %.f90

    $(FC) -c -o $@ $<

include kreis/module.mk

include quadrat/module.mk

include module.mk

prog: $(OBJ)

    $(FC) -o $@ $^
```

module.mk:

```
SRC += main.f90
```

kreis/module.mk:

```
SRC += kreis/kreis.f90 kreis/kreissegment.f90
```

quadrat/module.mk:

```
SRC += quadrat/quadrat.f90
```

Es gibt nur ein Makefile im Projekthauptverzeichnis. Sämtliche unterverzeichnisspezifischen Details (hier nur die Bekanntgabe der Quellcodedateien) werden in den jeweiligen Unterverzeichnissen in eigenen Include-Dateien (.mk) festgelegt und in das Makefile eingebunden. Da, anders als beim rekursiven Make, nicht in die einzelnen Unterverzeichnisse gewechselt wird, werden allfällige mod-Dateien auch nur in das Projekthauptverzeichnis (= das aktuelle Verzeichnis) geschrieben.

97.3.3. Weiterführende Make-Infos

- GNU MAKE MANUAL[1]
- ROBERT MECKLENBURG: MANAGING PROJECTS WITH GNU MAKE, O'REILLY OPENBOOK, 2004[2]
- PETER MILLER: RECURSIVE MAKE CONSIDERED HARMFUL, 1997[3]

97.4. CMake

CMake ist kein Make-Klon, sondern ein moderner Autotools-Ersatz.

Gleiches Beispiel wie bei HTTP://MW/INDEX.PHP5/MAKENICHTREKURSIVE_-MAKE-VARIANTE[4], es muss in diesem Fall nur eine

1 HTTP://WWW.GNU.ORG/SOFTWARE/MAKE/MANUAL/
2 HTTP://WWW.OREILLY.COM/CATALOG/MAKE3/BOOK/INDEX.CSP
3 HTTP://MILLER.EMU.ID.AU/PMILLER/BOOKS/RMCH/
4 HTTP://MW/INDEX.PHP5/MAKE#NICHTREKURSIVE_MAKE-{}VARIANTE

CMakeLists.txt-Datei im Projekthauptverzeichnis erstellt werden. Makefile und dazugehörende Dateien werden automatisch beim cmake-Aufruf erstellt.

CMakeLists.txt:

```
PROJECT(bsp Fortran)
SET(src
      main.f90
      kreis/kreis.f90
      kreis/kreissegment.f90
      quadrat/quadrat.f90
   )
ADD_EXECUTABLE(prog ${src})
```

Generieren der Makefiles, etc.:

- `FC=g95 cmake .`

```
-- Check for working Fortran compiler: /xyz/bin/g95
-- Check for working Fortran compiler: /xyz/bin/g95 -- works
-- Checking whether /xyz/bin/g95 supports Fortran 90
-- Checking whether /xyz/bin/g95 supports Fortran 90 -- yes
-- Configuring done
-- Generating done
-- Build files have been written to: /abc/projektverzeichnis
```

Mittels FC=... wird der zu verwendende Fortran-Compiler festgelegt. Wenn irgendein auf dem System installierter Fortran-Compiler verwendet werden soll, kann diese Vorgabe auch entfallen. In der CMakeLists.txt muss die Programmiersprache Fortran ausdrücklich aktiviert werden. Entweder wie hier im PROJECT-Statement oder alternativ auch über die ENABLE_LANGUAGE-Anweisung.

Programmerstellung:

- `make`

```
Scanning dependencies of target prog
[ 25%] Building Fortran object CMakeFiles/prog.dir/kreis/kreis.o
[ 50%] Building Fortran object
 CMakeFiles/prog.dir/kreis/kreissegment.o
[ 75%] Building Fortran object CMakeFiles/prog.dir/quadrat/quadrat.o
[100%] Building Fortran object CMakeFiles/prog.dir/main.o
Linking Fortran executable prog
```

CMake (2.4-patch 7) erkennt nur Dateien mit den Endungen f, F, ,f77, F77, f90, F90, for, f95 und F95, jedoch keine mit f03 oder F03.

CMake-Homepage:

- CMAKE CROSS-PLATFORM MAKE[5]

97.5. SCons

- SCONS[6]
- SCONS-HOMEPAGE[7]
- SCONS WIKI - MULTIPLEDIRECTORYFORTRANBUILD[8]

5 HTTP://WWW.CMAKE.ORG/HTML/INDEX.HTML
6 HTTP://DE.WIKIPEDIA.ORG/WIKI/SCONS
7 HTTP://WWW.SCONS.ORG/
8 HTTP://WWW.SCONS.ORG/WIKI/MULTIPLEDIRECTORYFORTRANBUILD?HIGHLIGHT=%28MULTIPLEDIRECTORYFORTRANBUILD%29

98. Photran - Eine IDE für Fortran basierend auf Eclipse

Für Fortran existieren bei weitem nicht so viele freie IDEs wie für C/C++. Eines der wenigen, wenn nicht sogar derzeit das einzige plattformübergreifende Werkzeug dieser Art ist Photran. Es basiert auf der bekannten Eclipse-IDE und deren C/C++-Erweiterung CDT. Photran weist aber noch nicht den Reife- und Fertigstellungsgrad seiner Java- oder C/C++-Pendants auf.

Leider existiert kein offizielles ausführliches Benutzerhandbuch für Photran und auch in diesem Buch wird momentan keine Photran-Kurzanleitung geboten. Grund ist, dass sich die Photran-Benutzerschnittstelle von Version zu Version noch sehr stark ändert. Es sollen hier also nur kurz einige Photran-Begriffe angerissen werden. Für weitere Informationen zu Photran wird auf die ehe spärliche offizielle Photran-Dokumentation verwiesen. Vieles in Photran ist ohnehin sehr ähnlich der Eclipse-IDE mit CDT-Erweiterung.

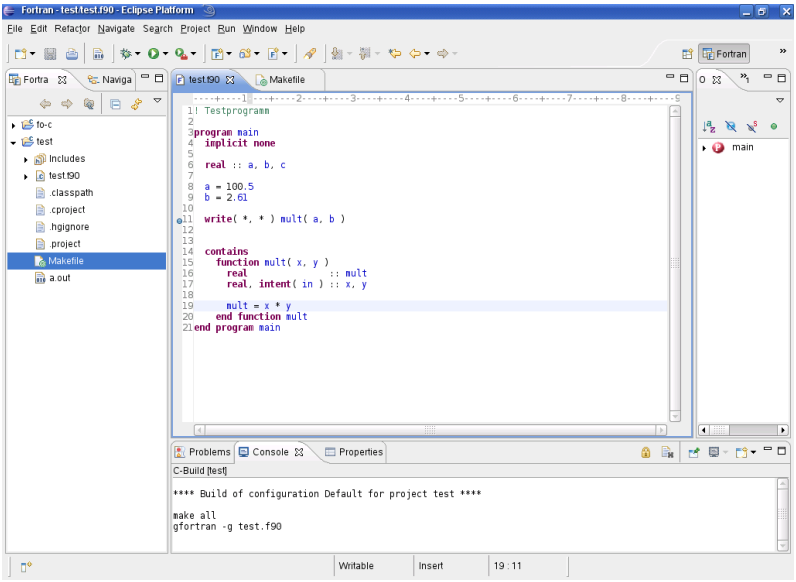


Abb. 71

Download-Varianten:

- Full Photran: Ein abgespecktes Eclipse mit den erforderlichen CDT- und Photran-Packages. Sofern auf dem Rechner eine installierte JRE oder ein JDK vorhanden ist, kann nach dem Download und Entpacken dieses Softwarepakets mit der Photran-IDE gearbeitet werden.
- Photran Feature: In dieser Download-Variante ist nur die Fortran-Erweiterung für Eclipse enthalten. Dieses Paket muss also in eine schon vorhandene Eclipse-Installation eingespielt werden. Auch das CDT ist in diesem Paket nicht inkludiert.

In älteren Photran-Versionen konnte die gewünschte Projektart ("Standard Make" oder "Managed Make") noch direkt bei der

Erstellung eines Photran-Projektes über Menüpunkte angewählt werden. Diese Möglichkeit ist in der Version 4.0 beta 2 entfallen. Die aktuell erforderlichen Schritte hin zu einem "Standard Make"- oder "Managed Make"-Projekt sind auf der Photran-Seite "Documentation for Photran" beschrieben.

- "Standard Make": Der Programmierer muss sich selbst um die Erstellung der für das Programm erforderlichen Makefiles kümmern.
- "Managed Make": Photran erstellt eigenständig die erforderlichen Makefiles. Dieser Projekttyp ist derzeit noch mit Vorsicht zu geniessen.

98.1. Weblinks

- PHOTRAN-WEBSITE[1]
- "DOCUMENTATION FOR PHOTRAN" AUF DER PHOTRAN-WEBSITE[2]

1 HTTP://WWW.ECLIPSE.ORG/PHOTRAN
2 HTTP://WWW.ECLIPSE.ORG/PHOTRAN/DOCUMENTATION.PHP

Teil XI.

Weblinks

98.2. Wikis

- Wikipedia zum Thema Fortran[3]
- Das englischsprachige Fortran-Wikibook[4]

98.3. Standards

- ISO/IEC JTC1/SC22/WG5 Offizielle Homepage für Fortran-Standards[5]
- J3 Fortran Standards Technical Committee[6]
- Der FORTRAN 66-Standard, ANSI USAS X3.9-1966[7]
- Der FORTRAN 77-Standard X3J3/90.4, ANSI X3.9-1978[8]
- Der Fortran 95-Standard J3/97-007, ISO/IEC 1539-1 : 1997 (Working Draft)[9]
- Der Fortran 2003-Standard J3/04-007, ISO/IEC 1539-1 : 2004 (Working Draft)[10]
- (Der Fortran 2008-Standard J3/07-007, Rev.3 vom 27. September 2007 (Working Draft))[11]

3 http://de.wikipedia.org/wiki/Fortran
4 http://en.wikibooks.org/wiki/Fortran
5 http://www.nag.co.uk/sc22wg5/
6 http://www.j3-{}fortran.org/
7 http://www.fh-{}jena.de/~{}kleine/history/languages/ansi-{}x3dot9-{}1966-{}Fortran66.pdf
8 http://www.fortran.com/F77_std/rjcnf.html
9 http://j3-{}fortran.org/doc/standing/archive/007/97-{}007r2/pdf/97-{}007r2.pdf
10 http://www.j3-{}fortran.org/doc/year/04/04-{}007.pdf
11 http://j3-{}fortran.org/doc/standing/links/007.pdf

98.4. Skripten, Tutorials, Bücher

98.4.1. Fortran 2003/2008

- The New Features of Fortran 2003 (John Reid)[12]

98.4.2. Fortran 90/95

- Programmieren in Fortran 90/95 (Gerd Groten, Forschungszentrum Jülich GmbH)[13]
- Programmieren in Fortran 90/95 (Dr. Heidrun Kolinsky, Rechenzentrum der Universität Bayreuth)[14]
- dmoz-Linksammlung für Fortran 90/95-Tutorials in englischer Sprache[15]
- Numerical Recipes in Fortran 90[16]
- Fortran 95-Fachwörterliste Englisch-Deutsch (RRZN Regionales Rechenzentrum für Niedersachsen)[17]

98.4.3. FORTRAN 77

- Professional Programmer's Guide to Fortran77 (Clive G. Page, University of Leicester)[18]

12 http://www.fortranplus.co.uk/resources/john_reid_new_2003.pdf
13 http://www.kfa-{}juelich.de/zam/docs/bhb/bhb_html/d0124/d0124.html
14 http://www.rz.uni-{}bayreuth.de/lehre/fortran90/vorlesung/index.html
15 http://dmoz.org/Computers/Programming/Languages/Fortran/Tutorials/Fortran_90_and_95/
16 http://www.nrbook.com/b/bookf90pdf.php
17 http://www.rrzn.uni-{}hannover.de/fileadmin/buecher/umdrucke/SPR.F95.2.pdf
18 http://www.star.le.ac.uk/~{}cgp/prof77.html

- Skriptum zu FORTRAN77 (S. Vetter, URZ Heidelberg)[19]
- Einführung in die FORTRAN Programmierung (Elsner, Universität Osnabrück)[20]
- dmoz-Linksammlung für FORTRAN 77-Tutorials in englischer Sprache[21]
- Numerical Recipes in Fortran 77[22]
- Sprachelemente in FORTRAN (Manfred Faber u. Gerhard Kahl, TU Wien)[23]

98.5. Compiler

- G95[24]
- GNU Fortran 95[25]
- Intel Fortran Compiler for Linux[26]
- Open64, The Open Research Compiler[27]
- Open Watcom[28]
- Silverfrost/Salford FTN95 Compiler [29]

19 http://web.urz.uni-{}heidelberg.de/Dokumentation/fortran77.html
20 http://www.rz.uni-{}osnabrueck.de/Zum_Nachlesen/Skripte_Tutorials/Programmieren_In_Fortran_77/pdf/fortran1.pdf
21 http://dmoz.org/Computers/Programming/Languages/Fortran/Tutorials/Fortran_77/
22 http://www.nrbook.com/b/bookfpdf.php
23 http://dollywood.itp.tuwien.ac.at/~{}edv2/Skripten/ftn.pdf
24 http://www.g95.org
25 http://gcc.gnu.org/fortran/
26 http://www.intel.com/cd/software/products/asmo-{}na/eng/compilers/flin/index.htm
27 http://www.open64.net
28 http://www.openwatcom.org
29 http://www.silverfrost.com/11/ftn95/overview.asp

- SUN STUDIO EXPRESS[30]

98.6. Debugger

- DER GNU DEBUGGER[31]

98.7. IDEs

- PHOTRAN[32]

98.8. Amüsantes

- TESTEN SIE IHREN PROGRAMMIERSTIL[33]
- REAL PROGRAMMERS DON'T USE PASCAL[34]

30 HTTP://DEVELOPERS.SUN.COM/SUNSTUDIO/DOWNLOADS/EXPRESS/INDEX.JSP
31 HTTP://DIRECTORY.FSF.ORG/GDB.HTML
32 HTTP://WWW.ECLIPSE.ORG/PHOTRAN/
33 HTTP://WWW.PDBM.DE/HUMOR/PROGRAMMIERSTIL.HTML
34 HTTP://WWW.EE.RYERSON.CA/~{}ELF/HACK/REALMEN.HTML

Teil XII.

Stichwortverzeichnis

- ** (Potenz-Operator)[35]
- +, -, *, /, ** (Arithmetische Operatoren, FORTRAN 77)[36]
- +, -, *, /, ** (Arithmetische Operatoren, Fortran 95)[37]
- = (Wertzuweisung, FORTRAN 77)[38]
- = (Wertzuweisung, Fortran 95)[39]
- <, <=, >, >=, ==, /= (Vergleichsoperatoren, Fortran 95)[40]

98.9. A

- abs (Fortran 95)[41]
- ABS (FORTRAN 77)[42]
- Absolutwert (FORTRAN 77)[43]
- Absolutwert (Fortran 95)[44]
- ACOS (FORTRAN 77)[45]

35 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%23Eigenschaften%20

36 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Arithmetische%20Operatoren%20

37 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Arithmetische%20Operatoren%20

38 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Wertzuweisung%20

39 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Wertzuweisung%20

40 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Vergleichsoperatoren%20

41 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Absolutwert%20

42 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Absolutwert%20

43 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Absolutwert%20

44 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Absolutwert%20

45 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Arkusfunktionen%20

- ADJUSTL (FORTRAN 95)[46]
- ADJUSTR (FORTRAN 95)[47]
- AIMAG (FORTRAN 95)[48]
- AIMAG (FORTRAN 77)[49]
- AINT (FORTRAN 95)[50]
- AINT (FORTRAN 77)[51]
- ALLOCATE (FORTRAN 95)[52]
- ALLOCATED (FORTRAN 95)[53]
- ALOG (FORTRAN 77)[54]
- ALOG10 (FORTRAN 77)[55]
- AMAX0 (FORTRAN 95)[56]
- AMAX0 (FORTRAN 77)[57]

---

46 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Sonstige%20

47 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Sonstige%20

48 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Komplexe%20Zahlen%20

49 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Komplexe%20Zahlen%20

50 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Rundung%20

51 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Rundung%20

52 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Speicherplatz%20dynamisch%20reservieren%20und%20freigeben%20

53 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Feldfunktionen%20

54 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Logarithmus%20naturalis%20

55 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Dekadischer%20Logarithmus%20

56 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Maximum%20

57 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Maximum%20

- AMAX1 (FORTRAN 95)[58]
- AMAX1 (FORTRAN 77)[59]
- AMIN0 (FORTRAN 95)[60]
- AMIN0 (FORTRAN 77)[61]
- AMIN1 (FORTRAN 95)[62]
- AMIN1 (FORTRAN 77)[63]
- AMOD (FORTRAN 95)[64]
- AMOD (FORTRAN 77)[65]
- .AND. (FORTRAN 77)[66]
- .AND. (FORTRAN 95)[67]
- ANINT (FORTRAN 95)[68]
- ANINT (FORTRAN 77)[69]

---

58 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Maximum%20

59 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Maximum%20

60 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Minimum%20

61 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Minimum%20

62 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Minimum%20

63 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Minimum%20

64 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Modulo%20

65 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Modulo%20

66 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Logische%20Operatoren%20

67 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Logische%20Operatoren%20

68 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Rundung%20

69 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Rundung%20

- ANY (FORTRAN 95)[70]
- ARITHMETISCHES IF (FORTRAN 77)[71]
- ARKUSFUNKTIONEN (FORTRAN 77)[72]
- ARKUSFUNKTIONEN (FORTRAN 95)[73]
- ARRAY CONSTRUCTOR (FORTRAN 95)[74]
- ASIN (FORTRAN 95)[75]
- ASIN (FORTRAN 77)[76]
- ASSIGN (FORTRAN 77)[77]
- ASSIGNED GOTO (FORTRAN 77)[78]
- ASSOCIATED (ZEIGERFUNKTION, FORTRAN 95)[79]
- ASSOCIATED (ASSOZIATIONSSTATUS, FORTRAN 95)[80]
- ASSOZIATIONSSTATUS (FORTRAN 95)[81]

---

70 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Feldfunktionen%20

71 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Arithmetisches%20IF%20

72 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Arkusfunktionen%20

73 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Arkusfunktionen%20

74 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Eindimensionale%20Felder%20%28array%20constructor%29%20

75 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Arkusfunktionen%20

76 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Arkusfunktionen%20

77 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23ASSIGN%20und%20Assigned%20GOTO%20

78 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23ASSIGN%20und%20Assigned%20GOTO%20

79 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Zeigerfunktionen%20

80 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Assoziationsstatus%20

81 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Assoziationsstatus%20

- ATAN (FORTRAN 95)[82]
- ATAN (FORTRAN 77)[83]
- ATAN2 (FORTRAN 95)[84]
- ATAN2 (FORTRAN 77)[85]
- ATLAS, AUTOMATICALLY TUNED LINEAR ALGEBRA SOFTWARE[86]

98.10. B

- BEDINGTES GOTO (FORTRAN 77)[87]
- BENANNTE KONSTANTEN (FORTRAN 77)[88]
- BENANNTE KONSTANTEN (FORTRAN 95)[89]
- BINÄRZAHL (FORTRAN 95)[90]
- BIT_SIZE (FORTRAN 95)[91]
- BITFUNKTIONEN (FORTRAN 95)[92]

82 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23ARKUSFUNKTIONEN%20

83 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2077%23ARKUSFUNKTIONEN%20

84 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23ARKUSFUNKTIONEN%20

85 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2077%23ARKUSFUNKTIONEN%20

86 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_BLAS_ATLAS%20

87 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2077%23BEDINGTES%20GOTO%20

88 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN_77%23BENANNTE%20KONSTANTEN%20

89 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN_95%23BENANNTE%20KONSTANTEN%20

90 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN_95%23DATENTYPEN%20

91 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23BITFUNKTIONEN%20

92 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23BITFUNKTIONEN%20

- BLAS, BASIC LINEAR ALGEBRA SUBPROGRAMS[93]
- BTEST (FORTRAN 95)[94]

98.11. C

- C[95]
- C (ANHANG A)[96]
- C (FORTRAN UND C)[97]
- CABS (FORTRAN 95)[98]
- CABS (FORTRAN 77)[99]
- CALL (FORTRAN 95)[100]
- CALL (FORTRAN 77)[101]
- CCOS (FORTRAN 77)[102]
- CEILING (FORTRAN 95)[103]

93 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_BLAS_ATLAS%20

94 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23BITFUNKTIONEN%20

95 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_EINLEITUNG%23EIGENSCHAFTEN%20

96 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_ANHANG_A%20

97 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN_UND_C%20

98 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23ABSOLUTWERT%20

99 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2077%23ABSOLUTWERT%20

100 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23SUBROUTINE%20

101 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2077%23SUBROUTINE%20

102 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2077%23WINKELFUNKTIONEN%20

103 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23RUNDUNG%20

- CEXP (FORTRAN 95)[104]
- CEXP (FORTRAN 77)[105]
- CHAR (FORTRAN 95)[106]
- CHAR (FORTRAN 77)[107]
- CHARACTER (FORTRAN 95)[108]
- CHARACTER (FORTRAN 77)[109]
- CLOG (FORTRAN 77)[110]
- CLOSE (FORTRAN 95)[111]
- CLOSE (FORTRAN 77)[112]
- CMPLX (FORTRAN 95)[113]
- CMPLX (FORTRAN 77)[114]
- COMMON (FORTRAN 77)[115]

---

104 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23EXPONENTIALFUNKTION%20

105 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2077%23EXPONENTIALFUNKTION%20

106 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23UMWANDLUNG%20IN%20CHARACTER%20

107 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2077%23UMWANDLUNG%20IN%20CHARACTER%20

108 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN_95%23ZEICHENKETTEN%20

109 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN_77%23ZEICHENKETTEN%20

110 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2077%23LOGARITHMUS%20NATURALIS%20

111 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23CLOSE%20

112 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2077%23CLOSE%20

113 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23UMWANDLUNG%20IN%20COMPLEX%20

114 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2077%23UMWANDLUNG%20IN%20COMPLEX%20

115 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN_77%23COMMON%20

- COMPILER[116]
- COMPLEX (FORTRAN 95)[117]
- COMPLEX (FORTRAN 77)[118]
- CONJ (FORTRAN 95)[119]
- CONJ (FORTRAN 77)[120]
- CONTINUE (FORTRAN 77)[121]
- COS (FORTRAN 95)[122]
- COS (FORTRAN 77)[123]
- COSH (FORTRAN 95)[124]
- COSH (FORTRAN 77)[125]
- CSHIFT (FORTRAN 95)[126]
- CSIN (FORTRAN 77)[127]

---

116 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%23Compiler%20

117 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Datentypen%20

118 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Arithmetische%20Datentypen%20

119 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Komplexe%20Zahlen%20

120 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Komplexe%20Zahlen%20

121 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23CONTINUE%20

122 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Winkelfunktion%20

123 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Winkelfunktionen%20

124 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Hyperbelfunktionen%20

125 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Hyperbelfunktionen%20

126 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Feldfunktionen%20

127 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Winkelfunktionen%20

- CSQRT (FORTRAN 95)[128]
- CSQRT (FORTRAN 77)[129]
- CYCLE (FORTRAN 95)[130]
- CYGWIN[131]

98.12. D

- DABS (FORTRAN 95)[132]
- DABS (FORTRAN 77)[133]
- DACOS (FORTRAN 77)[134]
- DASIN (FORTRAN 77)[135]
- DATA (FORTRAN 77)[136]
- DATAN (FORTRAN 77)[137]
- DATAN2 (FORTRAN 77)[138]

---

128 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Quadratwurzel%20

129 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Quadratwurzel%20

130 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23cycle%20

131 http://de.wikibooks.org/wiki/Fortran%3A_G95%23Unix-{}artige%20inkl.%20Cygwin%20und%20MacOSX%20

132 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Absolutwert%20

133 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Absolutwert%20

134 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Arkusfunktionen%20

135 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Arkusfunktionen%20

136 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23DATA%20

137 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Arkusfunktionen%20

138 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Arkusfunktionen%20

- Datei (FORTRAN 77)[139]
- Datei (Fortran 95)[140]
- Datensatz (FORTRAN 77)[141]
- Datentyp (FORTRAN 77)[142]
- Datentyp (Fortran 95)[143]
- Datentyp höherer Genauigkeit (Fortran 95)[144]
- Datentypumwandlung (FORTRAN 77)[145]
- Datenverbund (Fortran 95)[146]
- DBLE (FORTRAN 77)[147]
- DCOS (FORTRAN 77)[148]
- DCOSH (FORTRAN 77)[149]
- ddim (Fortran 95)[150]

---

139 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Datei%20

140 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Datei%20

141 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Datensatz%20

142 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Datentypen%20

143 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Datentypen%20

144 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Datentypen%20h%F6herer%20Genauigkeit%20

145 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Datentypumwandlung%20

146 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Datenverbund%20

147 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Umwandlung%20in%20DOUBLE%20

148 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Winkelfunktionen%20

149 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Hyperbelfunktionen%20

150 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Positive%20Differenz%20

- DDIM (FORTRAN 77)[151]
- DEALLOCATE (FORTRAN 95)[152]
- DEBUGGER (ANHANG C)[153]
- DEXP (FORTRAN 95)[154]
- DEXP (FORTRAN 77)[155]
- DIM (FORTRAN 95)[156]
- DIM (FORTRAN 77)[157]
- DINT (FORTRAN 95)[158]
- DINT (FORTRAN 77)[159]
- DLOG (FORTRAN 77)[160]
- DISLIN[161]
- DLOG10 (FORTRAN 77)[162]

151 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Positive%20Differenz%20
152 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Speicherplatz%20dynamisch%20reservieren%20und%20freigeben%20
153 http://de.wikibooks.org/wiki/Fortran%3A_Anhang_C%20
154 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Exponentialfunktion%20
155 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Exponentialfunktion%20
156 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Positive%20Differenz%20
157 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Positive%20Differenz%20
158 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Rundung%20
159 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Rundung%20
160 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Logarithmus%20naturalis%20
161 http://de.wikibooks.org/wiki/Fortran%3A_DISLIN%20
162 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Dekadischer%20Logarithmus%20

- DMAX1 (FORTRAN 95)[163]
- DMAX1 (FORTRAN 77)[164]
- DMIN1 (FORTRAN 95)[165]
- DMIN1 (FORTRAN 77)[166]
- DMOD (FORTRAN 95)[167]
- DMOD (FORTRAN 77)[168]
- DNINT (FORTRAN 95)[169]
- DNINT (FORTRAN 77)[170]
- DO-IF-EXIT (FORTRAN 95)[171]
- DO-LISTE (FORTRAN 95)[172]
- DO-LISTE (FORTRAN 77)[173]
- DO-SCHLEIFE (FORTRAN 77)[174]

163 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Maximum%20

164 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Maximum%20

165 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Minimum%20

166 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Minimum%20

167 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Modulo%20

168 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Modulo%20

169 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Rundung%20

170 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Rundung%20

171 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23do-{}if-{}exit%20

172 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Implizite%20do-{}Liste%20

173 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Implizite%20DO-{}Liste%20

174 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23DO-{}Schleifen%20

- DOT_PRODUCT (FORTRAN 95)[175]
- DOUBLE PRECISION (FORTRAN 95)[176]
- DOUBLE PRECISION (FORTRAN 77)[177]
- DO WHILE (FORTRAN 95)[178]
- DO-ZÄHLSCHLEIFE (FORTRAN 95)[179]
- DPROD (FORTRAN 77)[180]
- DSIGN (FORTRAN 95)[181]
- DSIGN (FORTRAN 77)[182]
- DSIN (FORTRAN 77)[183]
- DSINH (FORTRAN 77)[184]
- DSQRT (FORTRAN 95)[185]
- DSQRT (FORTRAN 77)[186]

175 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Feldfunktionen%20

176 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Datentypen%20

177 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Arithmetische%20Datentypen%20

178 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23do%20while%20

179 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Die%20do-{}Z%E4hlschleife%20

180 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Double%20Precision-{}Produkt%20

181 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Vorzeichentransfer%20

182 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Vorzeichentransfer%20

183 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Winkelfunktionen%20

184 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Hyperbelfunktionen%20

185 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Quadratwurzel%20

186 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Quadratwurzel%20

- DTAN (FORTRAN 77)[187]
- DTANH (FORTRAN 77)[188]
- Dynamische Speicherallokation (Fortran 95)[189]

98.13. E

- Ein- und Ausgabe (FORTRAN 77)[190]
- Ein- und Ausgabe (Fortran 95)[191]
- else-if (Fortran 95)[192]
- ENTRY (FORTRAN 77)[193]
- epsilon (Fortran 95)[194]
- .EQ. (FORTRAN 77)[195]
- .EQV. (FORTRAN 77)[196]
- .EQV. (Fortran 95)[197]

187 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Winkelfunktionen%20

188 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Hyperbelfunktionen%20

189 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Dynamische%20Speicherallokation%20

190 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Ein-{}%20und%20Ausgabe%20

191 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Ein-{}%20und%20Ausgabe%20

192 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23else-{}if%20

193 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23ENTRY%20

194 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Weitere%20Funktionen%20

195 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Vergleichsoperatoren%20

196 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Logische%20Operatoren%20

197 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Logische%20Operatoren%20

- Ergebnisdatentyp (FORTRAN 77)[198]
- Ergebnisdatentyp (Fortran 95)[199]
- exit (Fortran 95)[200]
- exp (Fortran 95)[201]
- EXP (FORTRAN 77)[202]
- Explizite Typzuweisung (FORTRAN 77)[203]
- Explizite Typzuweisung (Fortran 95)[204]
- Explizite Typumwandlung (FORTRAN 77)[205]
- Explizite Typumwandlung (Fortran 95)[206]
- exponent (Fortran 95)[207]
- Exponentialfunktion (FORTRAN 77)[208]
- Exponentialfunktion (Fortran 95)[209]

198 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Ergebnisdatentyp%20

199 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Ergebnisdatentyp%20

200 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23exit%20

201 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Exponentialfunktion%20

202 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Exponentialfunktion%20

203 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Explizite%20Typanweisung%20

204 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Explizite%20Typanweisung%20

205 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Explizite%20Typumwandlung%20

206 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Explizite%20Typumwandlung%20

207 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Weitere%20Funktionen%20

208 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Exponentialfunktion%20

209 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Exponentialfunktion%20

- EXTERNAL (FORTRAN 95)[210]
- EXTERNAL (FORTRAN 77)[211]

98.14. F

- F[212]
- F2C[213]
- FELD (FORTRAN 77)[214]
- FELD (FORTRAN 95)[215]
- FELD (PARAMETER, FORTRAN 95)[216]
- FELD (PARAMETER, FORTRAN 77)[217]
- FELDFUNKTION (FORTRAN 95)[218]
- FELDINITIALISIERUNG (FORTRAN 95)[219]
- FLOAT (FORTRAN 95)[220]

---

210 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Prozeduren%20als%20Parameter%20

211 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Prozeduren%20als%20Parameter%20

212 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%23Varianten%20

213 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%23%DCbersetzer%20

214 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Felder%20

215 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Felder%20

216 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Felder%20als%20Parameter%20

217 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Felder%20als%20Parameter%20

218 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Feldfunktionen%20

219 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Feldinitialisierung%20

220 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Umwandlung%20in%20REAL%20

- FLOAT (FORTRAN 77)[221]
- FLOOR (FORTRAN 95)[222]
- FORALL-SCHLEIFE (FORTRAN 95)[223]
- FORMAT (FORTRAN 95)[224]
- FORMAT (FORTRAN 77)[225]
- FORMATIERUNG (FORTRAN 77)[226]
- FORMATIERUNG (FORTRAN 95)[227]
- FORTRAN[228]
- FORTRAN I[229]
- FORTRAN II[230]
- FORTRAN IV[231]
- FORTRAN 66[232]

---

221 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Umwandlung%20in%20REAL%20

222 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Rundung%20

223 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23forall-{}Schleife%20

224 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23format%20

225 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23FORMAT%20

226 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23FORMAT%20

227 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23format%20

228 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%20

229 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%23Versionen%20

230 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%23Versionen%20

231 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%23Versionen%20

232 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%23Versionen%20

- FORTRAN 77[233]
- FORTRAN 77 (EINLEITUNG)[234]
- FORTRAN 90[235]
- FORTRAN 95[236]
- FORTRAN 95 (EINLEITUNG)[237]
- FORTRAN 2003 (EINLEITUNG)[238]
- FORTSETZUNGSZEILE (FORTRAN 77)[239]
- FREE SOURCE FORM (FORTRAN 95)[240]
- FREIES ZEILENFORMAT (FORTRAN 95)[241]
- FUNCTION (FORTRAN 95)[242]
- FUNCTION (FORTRAN 77)[243]
- FUNKTIONSANWEISUNG (FORTRAN 77)[244]

---

233 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%20

234 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%23Versionen%20

235 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%23Versionen%20

236 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%20

237 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%23Versionen%20

238 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%23Versionen%20

239 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Das%20Zeilenformat%20

240 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Beispiel%3A%20Hallo%20Welt%20

241 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Das%20Zeilenformat%20

242 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23function%20

243 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23FUNCTION%20

244 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Funktionsanweisung%20

98.15. G

- G95[245]
- .GE. (FORTRAN 77)[246]
- GETARG (FORTRAN 95)[247]
- GNU FORTRAN 95 (GFORTRAN)[248]
- GOTO (FORTRAN 77)[249]
- GOTO (ASSIGNED, FORTRAN 77)[250]
- GOTO (BEDINGT, FORTRAN 77)[251]
- GRAPHICAL USER INTERFACE (DISLIN)[252]
- GRAPHICAL USER INTERFACE (JAPI)[253]
- GRAPHICAL USER INTERFACE (PILIB)[254]
- .GT. (FORTRAN 77)[255]
- GTK+ (PILIB)[256]

245 http://de.wikibooks.org/wiki/Fortran%3A_G95%20
246 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Vergleichsoperatoren%20
247 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Kommandozeilenargumente%20
248 http://de.wikibooks.org/wiki/Fortran%3A_Gfortran%20
249 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23GOTO%20
250 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23ASSIGN%20und%20Assigned%20GOTO%20
251 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Bedingtes%20GOTO%20
252 http://de.wikibooks.org/wiki/Fortran%3A_DISLIN%20
253 http://de.wikibooks.org/wiki/Fortran%3A_Japi%20
254 http://de.wikibooks.org/wiki/Fortran%3A_Pilib%20
255 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Vergleichsoperatoren%20
256 http://de.wikibooks.org/wiki/Fortran%3A_Pilib%20

98.16. H

- Hauptprogramm (FORTRAN 77)[257]
- Hauptprogramm (Fortran 95)[258]
- Hexadezimalzahl (Fortran 95)[259]
- Hollerith (FORTRAN 77)[260]
- HPF, High Performance Fortran[261]
- Hyperbelfunktion (FORTRAN 77)[262]
- Hyperbelfunktionen (Fortran 95)[263]

98.17. I

- iabs (Fortran 95)[264]
- IABS (FORTRAN 77)[265]

---

257 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Die%20Programmstruktur%20f%FCr%20das%20Hauptprogramm%20

258 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Die%20Programmstruktur%20f%FCr%20das%20Hauptprogramm%20

259 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Datentypen%20

260 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Hollerith-{}Konstanten%20

261 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%23Varianten%20

262 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Hyperbelfunktionen%20

263 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Hyperbelfunktionen%20

264 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Absolutwert%20

265 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Absolutwert%20

- IAND (FORTRAN 95)[266]
- IARGC (FORTRAN 95)[267]
- IBCLR (FORTRAN 95)[268]
- IBSET (FORTRAN 95)[269]
- ICHAR (FORTRAN 95)[270]
- ICHAR (FORTRAN 77)[271]
- IDIM (FORTRAN 95)[272]
- IDIM (FORTRAN 77)[273]
- IDINT (FORTRAN 77)[274]
- IDNINT (FORTRAN 95)[275]
- IDNINT (FORTRAN 77)[276]
- IEOR (FORTRAN 95)[277]

266 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Bitfunktionen%20

267 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Kommandozeilenargumente%20

268 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Bitfunktionen%20

269 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Bitfunktionen%20

270 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Umwandlung%20in%20INTEGER%20

271 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Umwandlung%20in%20INTEGER%20

272 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Positive%20Differenz%20

273 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Positive%20Differenz%20

274 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Umwandlung%20in%20INTEGER%20

275 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Rundung%20

276 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Rundung%20

277 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Bitfunktionen%20

- IF (ARITHMETISCH, FORTRAN 77)[278]
- IF-VERWEIGUNG (FORTRAN 77)[279]
- IF-EINZEILER (FORTRAN 95)[280]
- IF-THEN (FORTRAN 95)[281]
- IF-THEN-ELSE (FORTRAN 95)[282]
- IFIX (FORTRAN 95)[283]
- IFIX (FORTRAN 77)[284]
- IMPLICIT (FORTRAN 95)[285]
- IMPLICIT (FORTRAN 77)[286]
- IMPLIZITE TYPUMWANDLUNG (FORTRAN 77)[287]
- IMPLIZITE TYPUMWANDLUNG (FORTRAN 95)[288]

278 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Arithmetisches%20IF%20

279 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23IF-{}Verzweigungen%20

280 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Der%20if-{}Einzeiler%20

281 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23if-{}then%20

282 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23if-{}then-{}else%20

283 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Umwandlung%20in%20INTEGER%20

284 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Umwandlung%20in%20INTEGER%20

285 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Implizite%20Typanweisung%20

286 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Implizite%20Typanweisung%20

287 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Implizite%20Typumwandlung%20bei%20Operanden%20gemischten%20Datentyps%20

288 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Implizite%20Typumwandlung%20bei%20Operanden%20gemischten%20Datentyps%20

- INCLUDE (FORTRAN 77)[289]
- INDEX (FORTRAN 95)[290]
- INDEX (FORTRAN 77)[291]
- INT (FORTRAN 95)[292]
- INT (FORTRAN 77)[293]
- INTEGER (FORTRAN 95)[294]
- INTEGER (FORTRAN 77)[295]
- INTEL FORTRAN COMPILER[296]
- INTENT (FORTRAN 95)[297]
- INTERFACE (FORTRAN UND C)[298]
- INTRINSIC (FORTRAN 95)[299]
- INTRINSIC (FORTRAN 77)[300]

289 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23INCLUDE%20

290 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Index%20eines%20Teilstrings%20

291 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Index%20eines%20Teilstrings%20

292 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Umwandlung%20in%20INTEGER%20

293 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%3A_Standardfunktionen%23Umwandlung_in_INTEGER%20

294 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Datentypen%20

295 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Arithmetische%20Datentypen%20

296 http://de.wikibooks.org/wiki/Fortran%3A_Ifort%20

297 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23subroutine%20

298 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_und_C%23Interface%20

299 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Prozeduren%20als%20Parameter%20

300 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Prozeduren%20als%20Parameter%20

- INTRINSIC FUNCTION (FORTRAN 77)[301]
- INTRINSIC FUNCTION (FORTRAN 95)[302]
- IOR (FORTRAN 95)[303]
- ISHFT (FORTRAN 95)[304]
- ISHFTC (FORTRAN 95)[305]
- ISIGN (FORTRAN 95)[306]
- ISIGN (FORTRAN 77)[307]

98.18. J

- JAPI[308]
- JAVA AWT (JAPI)[309]

301 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Standardfunktionen%20%28intrinsic%20functions%29%20
302 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Standardfunktionen%20%28intrinsic%20functions%29%20
303 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Bitfunktionen%20
304 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Bitfunktionen%20
305 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Bitfunktionen%20
306 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Vorzeichentransfer%20
307 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Vorzeichentransfer%20
308 http://de.wikibooks.org/wiki/Fortran%3A_Japi%20
309 http://de.wikibooks.org/wiki/Fortran%3A_Japi%20

98.19. K

- KIND (FORTRAN 95)[310]
- KIND (FUNKTION, FORTRAN 95)[311]
- KOMMANDOZEILENARGUMENT (FORTRAN 95)[312]
- KOMMENTARZEILE (FORTRAN 77)[313]
- KOMPLEXE ZAHLEN (FORTRAN 77)[314]
- KOMPLEXE ZAHLEN (FORTRAN 95)[315]

98.20. L

- LAPACK, LINEAR ALGEBRA PACKAGE[316]
- LBOUND (FORTRAN 95)[317]
- .LE. (FORTRAN 77)[318]
- LEN (FORTRAN 95)[319]

---

310 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Einfache%20Variante%20

311 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23kind-{}Parameter%20

312 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Kommandozeilenargumente%20

313 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Das%20Zeilenformat%20

314 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Komplexe%20Zahlen%20

315 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Komplexe%20Zahlen%20

316 http://de.wikibooks.org/wiki/Fortran%3A_LAPACK%20

317 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Feldfunktionen%20

318 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Vergleichsoperatoren%20

319 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Stringl%E4nge%20

- LEN (FORTRAN 77)[320]
- LEN_TRIM (FORTRAN 95)[321]
- LEXIKALISCHE FUNKTIONEN (FORTRAN 77)[322]
- LEXIKALISCHE FUNKTIONEN (FORTRAN 95)[323]
- LGE (FORTRAN 95)[324]
- LGE (FORTRAN 77)[325]
- LGT (FORTRAN 95)[326]* LGT (FORTRAN 77)[327]
- LINEARES GLEICHUNGSSYSTEM (LAPACK)[328]
- LLE (FORTRAN 95)[329]
- LLE (FORTRAN 77)[330]
- LLT (FORTRAN 95)[331]
- LLT (FORTRAN 77)[332]

---

320 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Stringl%E4nge%20

321 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Sonstige%20

322 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Lexikalische%20Funktionen%20

323 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Lexikalische%20Funktionen%20

324 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Lexikalische%20Funktionen%20

325 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Lexikalische%20Funktionen%20

326 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Lexikalische%20Funktionen%20

327 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Lexikalische%20Funktionen%20

328 http://de.wikibooks.org/wiki/Fortran%3A_LAPACK%20

329 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Lexikalische%20Funktionen%20

330 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Lexikalische%20Funktionen%20

331 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Lexikalische%20Funktionen%20

332 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Lexikalische%20Funktionen%20

- LOGARITHMUS (FORTRAN 95)[333]
- LOGARITHMUS (FORTRAN 77)[334]
- LOGICAL (FORTRAN 95)[335]
- LOGICAL (FORTRAN 77)[336]
- LOGISCHE OPERATOREN (FORTRAN 77)[337]
- .LT. (FORTRAN 77)[338]

98.21. M

- MACOSX[339]
- MATRIXOPERATION[340]
- MATRIZENRECHNUNG (BLAS UND ATLAS)[341]
- MATRIZENRECHNUNG (FORTRAN 95)[342]
- MAX (FORTRAN 95)[343]

---

333 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Logarithmus%20naturalis%20

334 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Logarithmus%20naturalis%20

335 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Logischer%20Datentyp%20

336 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Logischer%20Datentyp%20

337 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Logische%20Operatoren%20

338 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Vergleichsoperatoren%20

339 http://de.wikibooks.org/wiki/Fortran%3A_G95%23Unix-{}artige%20inkl.%20Cygwin%20und%20MacOSX%20

340 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%23Eigenschaften%20

341 http://de.wikibooks.org/wiki/Fortran%3A_BLAS_ATLAS%20

342 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Vektoren-{}%20und%20Matrizenrechnung%20

343 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Maximum%20

- MAX0 (FORTRAN 95)[344]
- MAX0 (FORTRAN 77)[345]
- MAX1 (FORTRAN 95)[346]
- MAX1 (FORTRAN 77)[347]
- MAXIMUM (FORTRAN 77)[348]
- MAXIMUM (FORTRAN 95)[349]
- MEHRDIMENSIONALES FELD (FORTRAN 77)[350]
- MEHRDIMENSIONALES FELD (FORTRAN 95)[351]
- MIN (FORTRAN 95)[352]
- MIN0 (FORTRAN 95)[353]
- MIN0 (FORTRAN 77)[354]
- MIN1 (FORTRAN 95)[355]

344 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Maximum%20

345 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Maximum%20

346 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Maximum%20

347 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Maximum%20

348 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Maximum%20

349 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Maximum%20

350 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Mehrdimensionale%20Felder%20

351 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Mehrdimensionale%20Felder%20

352 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Minimum%20

353 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Minimum%20

354 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Minimum%20

355 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Minimum%20

- MIN1 (FORTRAN 77)[356]
- Minimum (Fortran 95)[357]
- Minimum (FORTRAN 77)[358]
- mod (Fortran 95)[359]
- MOD (FORTRAN 77)[360]
- Modul (Fortran 95)[361]
- Modulo (FORTRAN 77)[362]
- Modulo (Fortran 95)[363]
- module (Fortran 95)[364]

98.22. N

- .NE. (FORTRAN 77)[365]
- .NEQV. (FORTRAN 77)[366]

---

356 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Minimum%20

357 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Minimum%20

358 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Minimum%20

359 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Modulo%20

360 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Modulo%20

361 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Module%20

362 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Modulo%20

363 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Modulo%20

364 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Module%20

365 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Vergleichsoperatoren%20

366 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Logische%20Operatoren%20

- .NEQV. (Fortran 95)[367]
- nint (Fortran 95)[368]
- NINT (FORTRAN 77)[369]
- .NOT. (FORTRAN 77)[370]
- .NOT. (Fortran 95)[371]
- not (Fortran 95)[372]
- null (Fortran 95)[373]
- nullify (Fortran 95)[374]

98.23. O

- Oktalzahl (Fortran 95)[375]
- open (Fortran 95)[376]
- OPEN (FORTRAN 77)[377]

---

367 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Logische%20Operatoren%20

368 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Rundung%20

369 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Rundung%20

370 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Logische%20Operatoren%20

371 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Logische%20Operatoren%20

372 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Bitfunktionen%20

373 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Assoziationsstatus%20

374 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Assoziationsstatus%20

375 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Datentypen%20

376 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23open%20

377 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23OPEN%20

- Operatorenpriorität (FORTRAN 77)[378]
- Operatorenpriorität (Fortran 95)[379]
- .OR. (FORTRAN 77)[380]
- .OR. (Fortran 95)[381]

98.24. P

- parameter (Fortran 95)[382]
- PARAMETER (FORTRAN 77)[383]
- PAUSE (FORTRAN 77)[384]
- pilib[385]
- pointer (Fortran 95)[386]
- Positive Differenz (FORTRAN 77)[387]
- Prozedur (Parameter, FORTRAN 77)[388]

---

378 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Operatorenpriorit%E4t%20

379 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Operatorenpriorit%E4t%20

380 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Logische%20Operatoren%20

381 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Logische%20Operatoren%20

382 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Benannte%20Konstanten%20

383 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Benannte%20Konstanten%20

384 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23PAUSE%20

385 http://de.wikibooks.org/wiki/Fortran%3A_Pilib%20

386 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Zeiger%20in%20Fortran%2095%20

387 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Positive%20Differenz%20

388 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Prozeduren%20als%20Parameter%20

- Prozedur (Parameter, Fortran 95)[389]

98.25. Q

- Quadratwurzel (FORTRAN 77)[390]

98.26. R

- Ratfor[391]
- read (Fortran 95)[392]
- READ (FORTRAN 77)[393]
- real (Fortran 95)[394]
- real (Funktion, Fortran 95)[395]
- REAL (FORTRAN 77)[396]
- REAL (Funktion, FORTRAN 77)[397]

---

389 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Prozeduren%20als%20Parameter%20

390 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Quadratwurzel%20

391 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%23Varianten%20

392 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23read%20

393 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23READ%20

394 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Datentypen%20

395 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Umwandlung%20in%20REAL%20

396 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Arithmetische%20Datentypen%20

397 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Umwandlung%20in%20REAL%20

- Rekursiver Unterprogrammaufruf (Fortran 95)[398]
- repeat (Fortran 95)[399]
- "Repeat Until"-Schleife (FORTRAN 77)[400]
- reshape (Fortran 95)[401]
- return (Fortran 95)[402]
- RETURN (FORTRAN 77)[403]
- Rundung (FORTRAN 77)[404]
- Rundung (Fortran 95)[405]

98.27. S

- SAVE (FORTRAN 77)[406]
- scan (Fortran 95)[407]
- Schleife (FORTRAN 77)[408]

---

398 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Rekursiver%20Unterprogrammaufruf%20

399 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Sonstige%20

400 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23%22Repeat%20Until%22-{}Schleife%20

401 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Mehrdimensionale%20Felder%20%28reshape%29%20

402 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23function%20

403 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23FUNCTION%20

404 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Rundung%20

405 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Rundung%20

406 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23SAVE%20

407 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Sonstige%20

408 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Verzweigungen%20und%20Schleifen%20

- SCHLEIFE (FORTRAN 95)[409]
- SELECT CASE (FORTRAN 95)[410]
- SELECTED_REAL_KIND (FORTRAN 95)[411]
- SEQUENTIELLE DATEI (FORTRAN 77)[412]
- SEQUENTIELLE DATEI (FORTRAN 95)[413]
- SIGN (FORTRAN 95)[414]
- SIGN (FORTRAN 77)[415]
- SIN (FORTRAN 95)[416]
- SIN (FORTRAN 77)[417]
- SINH (FORTRAN 95)[418]
- SINH (FORTRAN 77)[419]

409 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Verzweigungen%20und%20Schleifen%20

410 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23select%20case-{}Verzweigung%20

411 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23selected_real_kind%20

412 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Positionieren%20bei%20sequentiellen%20Dateien%20

413 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Positionieren%20bei%20sequentiellen%20Dateien%20

414 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Vorzeichentransfer%20

415 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Vorzeichentransfer%20

416 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Winkelfunktion%20

417 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Winkelfunktionen%20

418 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Hyperbelfunktionen%20

419 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Hyperbelfunktionen%20

- SIZE (FORTRAN 95)[420]
- SNGL (FORTRAN 95)[421]
- SNGL (FORTRAN 77)[422]
- SPALTENORGANISATION[423]
- SQRT (FORTRAN 95)[424]
- SQRT (FORTRAN 77)[425]
- STANDARDFUNKTION (FORTRAN 77)[426]
- STANDARDFUNKTION (FORTRAN 95)[427]
- STATISCHE SPEICHERALLOKATION (FORTRAN 95)[428]
- STOP (FORTRAN 95)[429]
- STOP (FORTRAN 77)[430]

420 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Feldfunktionen%20

421 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Umwandlung%20in%20REAL%20

422 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Umwandlung%20in%20REAL%20

423 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Das%20Zeilenformat%20

424 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Quadratwurzel%20

425 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Quadratwurzel%20

426 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Standardfunktionen%20%28intrinsic%20functions%29%20

427 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Standardfunktionen%20%28intrinsic%20functions%29%20

428 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Statische%20Speicherallokation%20

429 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23stop%20

430 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23STOP%20

- Stringverknüpfung (FORTRAN 77)[431]
- Stringverknüpfung (Fortran 95)[432]
- Stringfunktionen (FORTRAN 77)[433]
- Stringfunktionen (Fortran 95)[434]
- subroutine (Fortran 95)[435]
- SUBROUTINE (FORTRAN 77)[436]
- Symbolische Namen (FORTRAN 77)[437]
- Symbolische Namen (Fortran 95)[438]

98.28. T

- tan (Fortran 95)[439]
- TAN (FORTRAN 77)[440]
- tanh (Fortran 95)[441]

---

431 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Verkn%FCpfungsoperator%20

432 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Verkn%FCpfungsoperator%20

433 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Stringfunktionen%20

434 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Stringfunktionen%20

435 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23subroutine%20

436 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23SUBROUTINE%20

437 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Symbolische%20Namen%20

438 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Symbolische%20Namen%20

439 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Winkelfunktion%20

440 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Winkelfunktionen%20

441 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Hyperbelfunktionen%20

- TANH (FORTRAN 77)[442]
- TARGET (FORTRAN 95)[443]
- TCL/TK (FORTRAN UND TCL)[444]
- TEILFELD (FORTRAN 95)[445]
- TEILKETTE (FORTRAN 77)[446]
- TEILKETTE (FORTRAN 95)[447]
- TRANSPOSE (FORTRAN 95)[448]
- TRIM (FORTRAN 95)[449]
- TYPE (FORTRAN 95)[450]

98.29. U

- UBOUND (FORTRAN 95)[451]
- UNIT (FORTRAN 77)[452]

---

442 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Hyperbelfunktionen%20

443 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Zeiger%20in%20Fortran%2095%20

444 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_und_Tcl%20

445 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Teilfelder%20

446 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Teilketten%20

447 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Teilketten%20

448 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Vektoren-{}%20und%20Matrizenrechnung%20

449 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Sonstige%20

450 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Datenverbund%20

451 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Feldfunktionen%20

452 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23UNIT%20

- UNTERPROGRAMM (FORTRAN 77)[453]
- UNTERPROGRAMM (FORTRAN 95)[454]
- USE (FORTRAN 95)[455]

98.30. V

- VARIABLE (FORTRAN 77)[456]
- VARIABLE (FORTRAN 95)[457]
- VEKTOROPERATION[458]
- VEKTORRECHNUNG (BLAS UND ATLAS)[459]
- VEKTORRECHNUNG (FORTRAN 95)[460]
- VERGLEICHSOPERATOR (FORTRAN 77)[461]
- VERGLEICHSOPERATOR (FORTRAN 95)[462]
- VERIFY (FORTRAN 95)[463]

---

453 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Unterprogramme%20

454 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Unterprogramme%20

455 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Module%20

456 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Variablen%20

457 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Variablen%20

458 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%23Eigenschaften%20

459 http://de.wikibooks.org/wiki/Fortran%3A_BLAS_ATLAS%20

460 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Vektoren-{}%20und%20Matrizenrechnung%20

461 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN%2077%23Vergleichsoperatoren%20

462 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Vergleichsoperatoren%20

463 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Sonstige%20

- VERKETTETE LISTE (FORTRAN 95)[464]
- VERZWEIGUNG (FORTRAN 77)[465]
- VERZWEIGUNG (FORTRAN 95)[466]
- VORZEICHENTRANSFER (FORTRAN 77)[467]
- VORZEICHENTRANSFER (FORTRAN 95)[468]

98.31. W

- WHERE-ELSEWHERE-SCHLEIFE (FORTRAN 95)[469]
- WHERE-SCHLEIFE (FORTRAN 95)[470]
- "WHILE"-SCHLEIFE (FORTRAN 77)[471]
- WINKELFUNKTION (FORTRAN 77)[472]
- WINKELFUNKTION (FORTRAN 95)[473]
- WRITE (FORTRAN 95)[474]

---

464 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23VERKETTETE%20LISTEN%20

465 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2077%23VERZWEIGUNGEN%20UND%20SCHLEIFEN%20

466 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23VERZWEIGUNGEN%20UND%20SCHLEIFEN%20

467 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2077%23VORZEICHENTRANSFER%20

468 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23VORZEICHENTRANSFER%20

469 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23WHERE-{}ELSEWHERE%20

470 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23WHERE-{}SCHLEIFE%20

471 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2077%23%22WHILE%22-{}SCHLEIFE%20

472 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2077%23WINKELFUNKTIONEN%20

473 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN%2095%23WINKELFUNKTION%20

474 HTTP://DE.WIKIBOOKS.ORG/WIKI/FORTRAN%3A_FORTRAN_95%23WRITE%20

- WRITE (FORTRAN 77)[475]

98.32. X

98.33. Y

98.34. Z

- Zeichenvorrat (FORTRAN 77)[476]
- Zeichenvorrat (Fortran 95)[477]
- Zeiger (Fortran 95)[478]
- Zeigerfunktionen (Fortran 95)[479]
- Zeilenformat (Einleitung)[480]
- Zeilenformat (FORTRAN 77)[481]
- Zeilenformat (Fortran 95)[482]

475 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23WRITE%20

476 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Der%20FORTRAN-{}Zeichenvorrat%20

477 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Der%20Fortran-{}Zeichenvorrat%20

478 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Zeiger%20

479 http://de.wikibooks.org/wiki/Fortran%3A_Fortran%2095%23Zeigerfunktionen%20

480 http://de.wikibooks.org/wiki/Fortran%3A_Einleitung%23Versionen%20

481 http://de.wikibooks.org/wiki/Fortran%3A_FORTRAN_77%23Das%20Zeilenformat%20

482 http://de.wikibooks.org/wiki/Fortran%3A_Fortran_95%23Das%20Zeilenformat%20

99. Autoren

Edits	User
7	AKKORDEON[1]
2	COMMONSDELINKER[2]
31	DIRK HUENNIGER[3]
1	FILEPETER[4]
1	FORTRANPLUS[5]
1	HEFTIGINDENAFTER[6]
3	HEULER06[7]
391	INTRUDER[8]
3	JUETHO[9]

1 HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:AKKORDEON
2 HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:COMMONSDELINKER
3 HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:DIRK_HUENNIGER
4 HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:FILEPETER
5 HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:FORTRANPLUS
6 HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:HEFTIGINDENAFTER
7 HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:HEULER06
8 HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:INTRUDER
9 HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:JUETHO

2 KLAUS EIFERT[10]
1 LOEHDEN[11]
46 MICHAELFREY[12]
1 PHILIPENDULA[13]
1 SALATGURKE[14]
5 THEPACKER[15]
19 THORNARD[16]

10 HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:KLAUS_EIFERT
11 HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:LOEHDEN
12 HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:MICHAELFREY
13 HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:PHILIPENDULA
14 HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:SALATGURKE
15 HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:THEPACKER
16 HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:THORNARD

100. Bildnachweis

In der nachfolgenden Tabelle sind alle Bilder mit ihren Autoren und Lizenen aufgelistet.

Für die Namen der Lizenzen wurden folgende Abkürzungen verwendet:

- GFDL: Gnu Free Documentation License. Der Text dieser Lizenz ist in einem Kapitel diese Buches vollständig angegeben.
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. Der Text dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by-sa/3.0/ nachgelesen werden.
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. Der Text dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by-sa/2.5/ nachgelesen werden.
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. Der Text der englischen Version dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by-sa/2.0/ nachgelesen werden. Mit dieser Abkürzung sind jedoch auch die Versionen dieser Lizenz für andere Sprachen bezeichnet. Den an diesen Details interessierten Leser verweisen wir auf die Onlineversion dieses Buches.

- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. Der Text dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by-sa/1.0/ nachgelesen werden.

- cc-by-2.0: Creative Commons Attribution 2.0 License. Der Text der englischen Version dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by/2.0/ nachgelesen werden. Mit dieser Abkürzung sind jedoch auch die Versionen dieser Lizenz für andere Sprachen bezeichnet. Den an diesen Details interessierten Leser verweisen wir auf die Onlineversion dieses Buches.

- cc-by-2.0: Creative Commons Attribution 2.0 License. Der Text dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by/2.0/deed.en nachgelesen werden.

- cc-by-2.5: Creative Commons Attribution 2.5 License. Der Text dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by/2.5/deed.en nachgelesen werden. Mit dieser Abkürzung sind jedoch auch die Versionen dieser Lizenz für andere Sprachen bezeichnet. Den an diesen Details interessierten Leser verweisen wir auf die Onlineversion dieses Buches.

- cc-by-3.0: Creative Commons Attribution 3.0 License. Der Text dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by/3.0/deed.en nachgelesen werden. Mit dieser Abkürzung sind jedoch auch die Versionen dieser Lizenz für andere Sprachen bezeichnet. Den an diesen Details interessierten Leser verweisen wir auf die Onlineversion dieses Buches.

- GPL: GNU General Public License Version 2. Der Text dieser Lizenz kann auf der Webseite

http://www.gnu.org/licenses/gpl-2.0.txt nachgelesen werden.

- PD: This image is in the public domain. Dieses Bild ist gemeinfrei.

- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.

- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.

- LFK: Lizenz Freie Kunst. Der Text dieser Lizenz kann auf der Webseite http://artlibre.org/licence/lal/de nachgelesen werden.

- CFR: Copyright free use. Der Urheberrechtsinhaber erlaubt es jedem, dieses Bild für jeglichen Zweck, inklusive uneingeschränkter Weiterveröffentlichung, kommerziellem Gebrauch und Modifizierung, zu nutzen.

- EPL: Eclipse Public License. Der Text dieser Lizenz kann auf der Webseite http://www.eclipse.org/org/documents/epl-v10.php nachgelesen werden.

Bild	Autor	Lizenz
1		PD

2	=
	• SVG:
	  • commons: SHAZZ[1]
	  • pl.wiki: SHAZZ[2]
	• Bitmap project by:
	  • pl.wiki BORKOWSK[3] - W. Borkowski
	== {{int:filedesc
		GFDL

3	SURACHIT[4]	GFDL
4		PD
5		PD
6		PD
7		PD
8		PD
9		PD
10		PD
11	Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988.	PD
12	SIRIUSB[5]	GFDL
13	Arnold Reinhold	cc-by-sa-2.5
14		PD
15		PD
16	ART. LEBEDEV STUDIO[6] Original uploader was ORIGINALGAMER[7] at EN.WIKIPEDIA[8]	PD

1 HTTP://DE.WIKIBOOKS.ORG/WIKI/USER%3ASHAZZ
2 HTTP://DE.WIKIPEDIA.ORG/WIKI/PL%3AUSER%3ASHAZZ
3 HTTP://DE.WIKIPEDIA.ORG/WIKI/PL%3AUSER%3ABORKOWSK
4 HTTP://DE.WIKIBOOKS.ORG/WIKI/USER%3ASURACHIT
5 HTTP://DE.WIKIBOOKS.ORG/WIKI/USER%3ASIRIUSB
6 HTTP://WWW.ARTLEBEDEV.COM/
7 HTTP://DE.WIKIBOOKS.ORG/WIKI/%3AEN%3AUSER%3AORIGINALGAMER
8 HTTP://EN.WIKIPEDIA.ORG

17	RATOPI[9]	GFDL
18	User ARPINGSTONE[10] on EN.WIKIPEDIA[11]	PD
19	JAILBIRD[12]	cc-by-sa-2.0
20		PD
21	STEPHAN BAUM[13] (recolored by MZAJAC[14], converted to SVG by BOOYABAZOOKA[15])	GFDL
22		PD
23		PD
24		PD
25		PD
26		PD
27		PD
28		PD
29		PD
30		PD
31		PD
32	Leipnizkeks	GFDL
33		PD
34	User:Petwoe on German Wikipedia	GFDL
35	Intruder	GFDL
36		PD
37	Intruder	GFDL
38		PD
39		PD
40		PD
41		PD

9 HTTP://DE.WIKIBOOKS.ORG/WIKI/%3ADE%3AUSER%3ARATOPI
10 HTTP://DE.WIKIBOOKS.ORG/WIKI/%3AEN%3AUSER%3AARPINGSTONE
11 HTTP://EN.WIKIPEDIA.ORG
12 HTTP://DE.WIKIBOOKS.ORG/WIKI/USER%3AJAILBIRD
13 HTTP://DE.WIKIBOOKS.ORG/WIKI/USER%3ABAUMST
14 HTTP://DE.WIKIBOOKS.ORG/WIKI/%3AEN%3AUSER%3AMZAJAC
15 HTTP://DE.WIKIBOOKS.ORG/WIKI/USER%3ABOOYABAZOOKA

42		PD
43		PD
44		PD
45		PD
46		PD
47	MESSERWOLAND[16]	GFDL
48	Intruder	GFDL
49	Intruder	GFDL
50	Intruder	GFDL
51		PD
52		PD
53	Intruder	GFDL
54		PD
55		PD
56		PD
57		PD
58		PD
59		PD
60		PD
61		PD
62		PD
63		PD
64		PD
65		PD
66		PD
67		PD
68		PD
69		PD
70	MASA	GFDL
71	Thunderbolt No.999	EPL

16 HTTP://DE.WIKIBOOKS.ORG/WIKI/USER%3AMESSERWOLAND