Programmiersprachen: Fortran-9x, F und C

Überblick

Was früher einmal eine "höhere Programmiersprache" oder "Programmiersprache" schlechthin genannt wurde, nämlich eine Sprache auf leichter verständlichem und kompakterem Niveau als Assembler-Code, ist heute eigentlich schon in einem mittleren Bereich angesiedelt, da es inzwischen programmierbare Softwarepakete gibt, mit deren Hilfe sich komplexe Aufgaben, wie die Berechnung der Eigenwerte einer Matrix oder der Nullstellen eines Polynoms, per Knopfdruck (Mausklick!) lösen lassen. Trotzdem gehört das Erlernen einer oder mehrerer Programmiersprachen in diesem klassischen Sinn nach wie vor zur naturwissenschaftlichen Grundausbildung, da ja speziell in der Forschung kaum vorgefertigte Software zur Lösung aller Probleme zur Verfügung steht.

Programmiersprachen haben, ebenso wie Betriebssysteme und alle anderen Arten von Software, eine historische Entwicklung durchgemacht: Sie sind entstanden, waren in Mode, haben Religionskriege ausgelöst und sind wieder in Vergessenheit geraten—oder sie haben sich als von längerfristigem Wert erwiesen. Prinzipiell gibt es nicht "die" ideale Programmiersprache schlechthin, sondern die verschiedenen Sprachen sind jeweils für verschiedene Aufgaben mehr oder weniger gut geeignet, d.h. die verwendete Sprache sollte sich nach der Art der Anwendung richten. Die wichtigsten Programmiersprachen in den Naturwissenschaften sind in dieser Hinsicht wohl Fortran und C; da aber jede der beiden ihre typischen Vor- und Nachteile hat und sie einander eigentlich ergänzen, ist es vorteilhaft, sich Grundkenntnisse von *beiden* anzueignen.

Fortran-9x

Fortran ist in den 1950er Jahren entstanden und damit die älteste höhere Programmiersprache überhaupt. Für den Bau des ersten Fortran-I Compilers mußten so viele neue Konzepte und Optimierungsstrategien entwickelt werden, die später Allgemeingut der Compilertechnologie (auch für andere Sprachen) wurden, daß dieser Compiler zu den zehn wichtigsten Algorithmen des 20. Jahrhunderts gezählt wird. Wie schon der Name (FORmula TRANslator) sagt, ist Fortran optimal geeignet für numerische naturwissenschaftliche Berechnungen. Die Sprache zeichnet sich, was die Übertragung mathematischer Formeln in Programm-Befehle betrifft, durch eine intuitive Syntax aus, und die von guten Compilern erzeugten Codes sind äußerst effizient. Der Umfang der Sprache wurde laufend erweitert und in internationalen Standards festgelegt. Die derzeit gültigen "Dialekte" sind Fortran-90 bzw. 95, doch wird auf Rückwärtskompatibilität großer Wert gelegt, sodaß z.B. in F66 oder F77 (also in den 1960er oder 1970er Jahren) geschriebene Programme im Prinzip nach wie vor lauffähig sind. Ebenso ist, da kaum Wechselwirkungen mit anderen Systemprogrammen bestehen, Fortran eine der portabelsten Sprachen, d.h. ein auf dem Computer X entwickeltes Programm sollte praktisch ohne Änderungen auch auf dem Computer Y einsetzbar sein. Erst in F9x, also relativ spät und nur in Ansätzen, wurden "modernere" Konzepte wie Objektorientierung aus anderen Sprachen übernommen. Dafür sind in F9x gewisse Vektor- und Matrixoperationen Grundbestandteil der Sprache, ebenso wie Konstrukte zur Formulierung der Parallelisierbarkeit von Algorithmen.

F

Die Rückwärtskompatibilität stellt aber auch einen der größten Nachteile von Fortran dar. Dadurch, daß noch immer alles erlaubt und möglich sein soll, was in der Vergangenheit jemals Bestandteil von Fortran war, ist der Sprachumfang sehr groß, und es gibt viele Redundanzen, die besonders Anfänger verwirren. Außerdem kann man in Fortran sowohl elegante und effiziente als auch unlesbare und fehleranfällige Programme schreiben. Daher wurde, speziell für pädagogische Zwecke, die Sprache (=Untermenge von Fortran-90) "F" definiert. F ist sozusagen F90 ohne den historischen Ballast, d.h. jedes F-Programm ist immer ein gültiges F90- oder F95-Programm, aber es dürfen nur die moderneren Konstrukte von Fortran-90 verwendet werden. Darüber hinaus wird man generell zu einem sichereren Programmierstil gezwungen.

C

Die Programmiersprache C wurde um 1970 in Zusammenhang mit dem Betriebssystem Unix entwickelt. C ist, ebenso wie Fortran, eine sonst sehr allgemein einsetzbare Sprache, zeichnet sich aber vor allem durch die enge Beziehung zum Betriebssystem aus (der Großteil von Unix ist ja in C geschrieben). Da insbesondere mehr oder weniger alle Funktionen des Betriebssystems direkt aufgerufen werden können, ist es nicht schwer, aus einem C-Programm heraus direkt mit dem Betriebssystem, Netzwerkdiensten oder anderen Applikationen in Wechselwirkung zu treten. Daher sind auch fast alle größeren Softwarepakete in C geschrieben. Dazu kommt, daß die Syntax vieler anderer populärer Sprachen, wie C++, Perl oder Java, der von C sehr ähnlich ist und Grundkenntnisse von C auch das Erlernen modernerer Sprachen wesentlich erleichtern. Viele Eigenschaften von C können sich sowohl als Vor- als auch als Nachteil erweisen: So ist C bedeutend flexibler, und manches läßt sich wesentlich kompakter (und kryptischer) ausdrücken als in Fortran; gerade diese "Mächtigkeit" von C macht die Programme aber viel anfälliger für Programmierfehler. Andererseits können Hardware- und Betriebssystem-nahe Programme natürlich nur bedingt portabel sein. Was bei naturwissenschaftlichen Anwendungen in C oft stört, ist die Umständlichkeit mancher numerischen Ausdrücke. Bis vor kurzem kam dazu noch die Tatsache, daß Fortran-Compiler wesentlich effizienteren numerischen Code erzeugten, doch dürfte nach großen Anstrengungen im C-Compilerbau dieses Argument bald der Vergangenheit angehören.

In der folgenden Gegenüberstellung werden die wesentlichsten Sprachelemente und Konstruktionen von F (stellvertretend für F9x) und C behandelt. Das kann und soll keinen regulären Programmierkurs ersetzen—denn Programmieren ist in erster Linie Übungssache—, vielmehr soll gezeigt werden, daß es nicht schwer ist, eine weitere Programmiersprache zu erlernen, wenn man mit den Grundkonzepten (Schleifen, Verzweigungen, Unterprogramme usw.) in *einer* Sprache vertraut ist. In diesem Sinn werden nicht Gegensätze, sondern Gemeinsamkeiten zwischen F und C betont, d.h. es wird z.B. nicht "typischer" C-Programmierstil gezeigt, sondern Syntax und Beispiele werden so präsentiert, daß F und C möglichst ähnlich erscheinen.

Struktur eines F/C-Programms

Größere Anwendungen bestehen meist aus einem Haupt- und zahlreichen Unterprogrammen (u.U. sogar in verschiedenen Programmiersprachen), deren Source Code in einzelnen Files oder gruppenweise in Files gespeichert wird, die separat compiliert und dann gelinkt werden. Wir betrachten zunächst ausschließlich den einfacheren Fall, daß Haupt- und eventuelle Unterprogramme sich alle in ein und demselben File befinden, denn dann ist, wie die folgende Gegenüberstellung zeigt, der Aufbau von F- und C-Programmen im wesentlichen identisch. (Der allgemeinere Fall separat compilierter Files wird später behandelt.)

F-Programm:

Globale Vereinbarungen (in `module`*-Blöcken)*
  globale Variablen, Unterprogramme
Hauptprogramm (`program`*)*
  lokale Vereinbarungen
  ausführbare Befehle (Zuweisungen, Unterprogrammaufrufe)

C-Programm:

Preprocessor-Direktiven
Globale Vereinbarungen
  globale Variablen, Unterprogramme
Hauptprogramm (Funktion `main`*)*
  lokale Vereinbarungen
  ausführbare Befehle (Zuweisungen, Funktionsaufrufe)

Die ausführbaren Befehle werden in beiden Sprachen im Prinzip sequentiell, d.h. der Reihe nach, bzw. durch Schleifen, Verzweigungen usw. gesteuert, abgearbeitet. F9x kennt darüber hinaus auch noch parallele Konstruktionen.

Ein augenfälliger Unterschied sind die mit dem `#`-Zeichen beginnenden Preprocessor-Direktiven, die es nur in C gibt und die typischerweise am Beginn, aber auch an beliebigen anderen Stellen des Programms auftreten können. Der Preprocessor ist ein Filter, den das Programm passiert, bevor es an den eigentlichen Compiler weitergegeben wird. Mit Hilfe von Preprocessor-Direktiven können Konstante und Macros definiert, Teile des Programms maschinenspezifisch ein- oder ausgeblendet, zusätzlicher Code inkludiert werden usw.

Jedes F- oder C-Programm muß genau ein Hauptprogramm enthalten. Der Beginn desselben ist der sogenannte Entry Point, an dem die Ausführung des Programms beginnt, wenn es in den Computer geladen wird, und der natürlich eindeutig sein muß. In F wird der Beginn des Hauptprogramms durch das Keyword `program` markiert; in C, wo nicht zwischen Haupt- und Unterprogrammen unterschieden wird, durch den Beginn einer Funktion mit dem reservierten Namen `main`.

Sowohl in F als auch in C können Vereinbarungen, Anweisungen usw. im "freien Format" geschrieben werden, d.h. an beliebiger Stelle in einer Zeile. In C muß jede Variablendeklaration oder Anweisung durch einen Strichpunkt abgeschlossen werden, z.B. weist

```
x=y+z;
```

`x` die Summe von `y` und `z` zu. In F lautet dieselbe Anweisung einfach

```
x=y+z
```

Logisch zusammengehörende Befehle, sogenannte Blöcke, werden in F durch *keyword*... `end` *keyword* zusammengefaßt, in C durch Paare geschlungener Klammern, {...}. So werden z.B. Beginn und Ende des Hauptprogramms in F folgendermaßen gekennzeichnet

```
program name
  lokale Vereinbarungen
  ausführbare Befehle
  ...
end program name
```

in C aber durch

```
int main(...) {
  lokale Vereinbarungen
  ausführbare Befehle
  ...
}
```

Dabei muß in F der (Programm-)Name in den Zeilen `program` und `end program` übereinstimmen. Eine Besonderheit von C ist, daß an die Funktion `main` aus dem Betriebssystem die Kommandozeilen-Argumente als Parameter übergeben werden können.

Die logischen Blöcke werden meist durch Einrücken sichtbar gemacht. Um wieviel eingerückt wird und ob in C die geschlungenen Klammern in eigenen Zeilen stehen, ist Sache des persönlichen Geschmacks.

Variablendeklarationen und Datentypen

Variablennamen

Die Namen von Variablen, Prozeduren usw. können in beiden Sprachen beliebig lang sein und aus Buchstaben, Ziffern und gewissen Sonderzeichen bestehen. Bei zu langen Namen werden u.U. intern nur die ersten 31 Zeichen verwendet. Namen müssen mit einem Buchstaben beginnen, und in F ist als Sonderzeichen nur der Underscore "_" erlaubt, der aber nicht am Ende eines Namens stehen darf. In C ist die Groß- und Kleinschreibung von Namen signifikant, d.h. zwei Variablen, die sich nur durch die Groß- oder Kleinschreibung eines Buchstabens unterscheiden, gelten als verschieden. In F9x werden wegen der Rückwärtskompatibilität vom Compiler intern alle Namen in Großschreibung umgewandelt. Daher dürfen in F aus Gründen der Eindeutigkeit keine Variablen verwendet werden, die sich nur durch die Groß- bzw. Kleinschreibung der Buchstaben unterscheiden. Außerdem müssen in F die Schlüsselwörter der Sprache klein geschrieben werden.

Primitive Datentypen

Beide Sprachen kennen jeweils eine Reihe von Datentypen für Integer- und Floating Point-Zahlen sowie die Typen Character und Pointer. Die wichtigsten sind

F	C

`integer`	`byte`
	`int`
	`long`

`real`	`float` `double`
`complex`	
`character`	`char`
`logical`	
`pointer`	`*`

Die Default-Längen (2, 4 oder 8 Bytes) der Integer- und Floating Point (`real`-)Zahlen sind implementationsabhängig. In F und F9x gibt es aber die Möglichkeit, den gewünschten Datentyp über eine numerische Mindestpräzision maschinenunabhängig festzulegen. Für naturwissenschaftlich-technische Anwendung ist das Fehlen komplexer Zahlen in C manchmal störend.

Literals

Die Schreibweise für numerische und Character-Konstante ist in F und C praktisch dieselbe. So sind z.B.

`17`, `+58` oder `-3`

ganze und

`238.4` oder `-1.4708`

gültige Floating Point-Zahlen. Zur Darstellung sehr großer oder kleiner Werte gibt es auch noch die wissenschaftliche Notation

```
6.022e+23
```

wobei `e`$\pm$`n` für einen Faktor $10^{\pm n}$ steht. Komplexe Werte, die allerdings nur in F zur Verfügung stehen, werden als Paare von Real- und Imaginärteilen in runden Klammern geschrieben, also z.B.

```
(0.0,1.0)
```

für die imaginäre Einheit i.

Zeichenketten-Konstanten werden in beiden Sprachen zwischen doppelte Hochkommas gesetzt

```
"Das ist ein Text."
```

Einer der subtilen Unterschiede zwischen F und C besteht allerdings darin, daß in Fortran eine primitive Character-Variable, wenn sie entsprechen definiert wird, einen beliebig langen String enthalten kann, in C aber nur ein einzelnes Zeichen, das als Literal zwischen einfachen Hochkommas steht, also z.B.

```
'a'
```

Strings aus mehr als einem Zeichen sind in C stets Arrays.

Die Schreibweise der zwei möglichen logischen Werte, "wahr" und "falsch", in F ist

```
.true.
```

bzw.

```
.false.
```

(also mit je einem Punkt vor und hinter dem Wert).

Variablendeklarationen

In F9x gilt aus Gründen der Rückwärtskompatibilität zu älteren Sprachdialekten, daß der Typ von Variablen implizit durch den Anfangsbuchstaben festgelegt ist: Variablen, deren Name mit `i`–`n` beginnt, sind vom Typ `integer`, alle anderen (deren Name also mit `a`–`h` oder `o`–`z` beginnt) vom Typ `real`. Um Tipp- und Programmierfehler leichter zu finden, empfiehlt es sich allerdings, diese Konvention durch eine `implicit none`-Direktive am Beginn des Programms außer Kraft zu setzen. In F und C müssen ohnehin alle Variablen explizit deklariert werden.

Primitive Variablen deklariert man im einfachsten Fall auf folgende Weise

```
      F                C

integer::i,j,k     int i,j,k;
real::x,y,z        float x,y,z;
```

wozu im Fall von F noch Attribute kommen können. So definiert z.B.

```
integer,parameter::imax=100
```

keine Variable, sondern eine benannte Konstante, d.h. in diesem Fall, daß vom Compiler überall im Programm der Name `imax` durch den Wert `100` ersetzt wird. (In C würde man einen solchen "Parameter" typischerweise durch eine Preprocessor-Direktive

```
#define IMAX 100
```

definieren.)

Ein- und Ausgabe auf Tastatur und Schirm

Ausgabe

Der Aufbau der Befehle für die Ein- und Ausgabe ist in F und C vollkommen analog. Außerdem kann in F dieselbe Art von Befehlen sowohl für die Ein- und Ausgabe auf Files als auch für die Kommunikation mit Tastatur und Bildschirm verwendet werden (in C bestehen geringfügige Unterschiede).

Die Ausgabe von Daten aus einem F-Programm auf den Schirm erfolgt mit Hilfe des `write`-Befehls

`write(unit=*,fmt="(`*format*`)")` *expression_list*

Dabei wird mit dem Schlüsselwort `unit` im allgemeinen Fall der "Kommunikationskanal" (ein ganzzahliger Ausdruck, dessen Wert z.B. mit einem File assoziiert sein kann) angegeben. Der Kanal `"*"` steht für Ausgabe auf den Schirm. Das zweite Argument `fmt` enthält in einer Zeichenkette in runden Klammern eine Beschreibung des Formats, in dem die Ausgabe erfolgen soll. Legt man keinen Wert auf eine bestimmte Formatierung, so kann man sie mit Hilfe von `fmt=*` auch dem Compiler überlassen. *expression_list* gibt schließlich in einer durch Beistriche getrennten Liste von Ausdrücken die eigentlich auszugebenden Daten an.

Für die Ausgabe aus einem C-Programm auf den Schirm benützt man die `printf`-Funktion

`printf("`*format*`",`*expression_list*`);`

wobei *format* und *expression_list* wieder die Formatbeschreibung (allerdings ohne runde Klammern) und die Liste von auszugebenden Daten sind. In C fehlt leider das praktische `"*"`-Format.

Formatbeschreibung

In F besteht die Formatbeschreibung aus einer durch Beistriche getrennten Liste von Einzelformatbeschreibungen (oder Klartext), die angeben, wie das zugehörige Element der Datenliste dargestellt werden soll. Die wichtigsten Einzelformate sind:

`i`n	ganze Zahl mit n Stellen
`f`n`.`m	Kommazahl mit n Stellen insgesamt, davon m nach dem Komma
`es`n`.`m	Exponentialformat mit n Stellen insgesamt, davon m nach dem Komma
`a`n, `a`	Zeichenkette mit n Stellen bzw. angepaßter Länge
`tr`n	n Leerstellen
`/`	neue Zeile
`""`*text*`""`	*text* wird wörtlich ausgegeben

Einzelformate können durch Klammern in Gruppen zusammengefaßt und mit Wiederholungsfaktoren versehen werden, z.B.

```
      3i5  =  i5,i5,i5
2(tr1,f8.3)  =  tr1,f8.3,tr1,f8.3
```

In C wird der Text in der Formatbeschreibung *format* wörtlich ausgegeben, ausgenommen Einzelformatbeschreibungen, die mit dem `"%"`-Zeichen beginnen und wieder den Elementen in der Datenliste entsprechen

`%`n`d`	ganze Zahl mit n Stellen
`%`n`ld`	ganze Zahl vom Typ `long` mit n Stellen
`%`n`.`m`f`	Kommazahl mit n Stellen insgesamt, davon m nach dem Komma
`%`n`.`m`e`	Exponentialformat mit n Stellen insgesamt, davon m nach dem Komma
`%`n`.`m`g`	äquivalent `f` oder `e`, je nach Größe der Zahl
`%`n`s`, `%s`	Zeichenkette mit n Stellen bzw. angepaßter Länge
`\n`	neue Zeile

Da man jedenfalls den Typ der auszugebenden Ausdrücke spezifizieren muß, gibt es zwar kein Äquivalent zum `"*"`-Format von F, aber es genügt als Abkürzung auch schon die Angabe `%d`, `%f`, usw.

Ein wichtiger Unterschied zwischen dem `write`- und dem `printf`-Befehl besteht darin, daß in F am Ende jeder formatierten Ausgabe automatisch eine neue Zeile begonnen wird. Will man das z.B. in einer Eingabeaufforderung unterdrücken, kann man die `advance="no"`-Option verwenden. So druckt etwa

```
write(unit=*,fmt="("" a="")",advance="no")
```

den Text

```
a=
```

auf den Bildschirm und bleibt dann mit dem Cursor hinter dem Gleichheitszeichen stehen. Im Gegensatz dazu muß in C der Zeilenvorschub am Ende einer Zeile explizit mit Hilfe von `\n` angegeben werden. Soll also z.B. der Wert der `float`-Variable `x` ausgegeben und dann (für die nächste Ausgabe) eine neue Zeile begonnen werden, kann man das mit

```
printf("x=%f\n",x);
```

erreichen.

Eingabe

Zum Einlesen von Werten von der Tastatur gibt es die der Ausgabe analogen Befehle

`read(unit=*,fmt="(`*format*`)")` *variable_list*

bzw.

`scanf("`*format*`",`*variable_list*`);`

Dabei ist *variable_list* eine durch Beistriche getrennte Liste von Variablen, denen die eingegebenen Werte zugewiesen werden.

Da man sich bei der Eingabe von der Tastatur im allgemeinen an keine starre Formatierung binden möchte, gibt es in F auch für den `read`-Befehl das `"*"`-Format. In diesem Fall können die einzulesenden Werte beliebig eingegeben werden; es muß nur jeweils ein Wert vom nächsten durch ein geeignetes Trennzeichen (meist ein Beistrich) unterschieden werden. Sollen also z.B. den drei (entsprechend deklarierten) `real`-Variablen `x`, `y` und `z` die Werte `1.25`, `201.5` und `-33.429` zugewiesen werden, so kann das durch den Befehl

```
read(unit=*,fmt=*) x,y,z
```

und die Eingabe von

```
1.25,201.5,-33.429
```

von der Tastatur erfolgen. In C würde man dieselbe Eingabe mit

```
scanf("%f,%f,%f",&x,&y,&z);
```

in die `float`-Variablen `x`, `y` und `z` einlesen.

Dieses Beispiel illustriert auch einen der gravierendsten Unterschiede zwischen F und C: Während in F Parameter an Prozeduren (wie die Ein- und Ausgabebefehle) per Adresse übergeben werden, erfolgt in C die Parameterübergabe per Wert (s. später). Das hat zur Folge, daß bei einer Eingabe, wodurch ja der Wert einer Variablen verändert werden soll, die *Adresse* der Variable übergeben werden muß. Dazu wird in C dem Namen der Variable das `"&"`-Zeichen vorangestellt.

Steuerung des Programmflusses

Wenn nicht anders angegeben, werden die ausführbaren Befehle eines Programms der Reihe nach abgearbeitet. (F9x kennt allerdings auch Element der parallelen Verarbeitung.) Ein davon abweichender Programmfluß kann durch die Verwendung verschiedener Steuerungskonstruktionen erreicht werden. Die wichtigsten sind Wiederholung (Schleifen), bedingte Ausführung von Programmteilen (logische Verzweigungen) und Unterprogramme.

Schleifen

Die einfachste Form einer Schleife (`do`-Schleife) in F ist

[*label*`:`] `do` *count*`=`*start*`,` *end* [`,` *increment*]
  body
  ...
`end do` [*label*]

Dabei ist der Schleifenzähler *count* eine ganzzahlige Variable, die mit *start* initialisiert und zu der nach jedem Schleifendurchlauf *increment* addiert wird. Der Block von Befehlen (*body*) zwischen `do` und `end do` wird so oft ausgeführt als der Schleifenzähler *end* nicht überschritten wird (bzw. unterschritten wird, wenn *increment* negativ ist). Ist diese Abbruchbedingung schon vor dem ersten Schleifendurchlauf erfüllt, wird die Schleife überhaupt nicht ausgeführt. Wenn *increment* nicht angegeben ist, wird *count* bei jedem Durchlauf um 1 erhöht. *start*, *end* und *increment* können ganzzahlige Ausdrücke sein. Um bei längeren oder geschachtelten Schleifen Beginn und Ende leichter identifizieren zu können, kann der Schleifenbeginn mit einem Namen *label* markiert werden, der dann aber auch beim schließenden `end do` angegeben werden muß.

Aus einer `do`-Schleife springt man mit dem Befehl

`exit` [*label*]

Die Ausführung des Programms wird dann mit dem auf das Schleifenende folgenden Befehl fortgesetzt. Der Befehl

`cycle` [*label*]

hingegen bewirkt, daß nur der momentane Schleifendurchlauf abgebrochen und die Schleife mit der nächsten Iteration fortgesetzt wird. Durch Angabe von *label* kann man auch aus bzw. an das Ende von (außerhalb liegenden) geschachtelten Schleifen springen.

Eine spezielle Form der Schleife ist die Endlosschleife

[*label*:] `do`
  body
  ...
`end do` [*label*]

Natürlich muß eine Endlosschleife irgendwann mit Hilfe von `exit` verlassen werden.

Die analogen Konstruktionen in C sind `for`-Schleifen. Ihre allgemeine Form ist

`for(`*init_expr*`;`*test_expr*`;`*incrmt_expr*`) {`
  body
  ...
`}`

Im Gegensatz zu F ist man hier nicht an die Verwendung von expliziten Schleifenzählern gebunden, sondern *init_expr*, *test_expr* und *incrmt_expr* können Ausdrücke oder Zuweisungen allgemeiner Art sein. Der Initialisierungsausdruck *init_expr* (meist eine Zuweisung) wird vor Beginn der Schleife durchgeführt, der Abbruchausdruck *test_expr* (meist ein logischer Test) vor jedem Schleifendurchlauf und der Iterationsausdruck *incrmt_expr* (i.a. wieder eine Zuweisung) nach jedem Schleifendurchlauf. Bei Bedarf dürfen einer oder mehrere dieser Ausdrücke auch weggelassen werden. So kann man z.B. mit

```
for(;;) {
  body
  ...
}
```

eine Endlosschleife erzeugen.

Aus einer `for`-Schleife springt man mit

```
break;
```

Durch den Befehl

```
continue;
```

wird nur der momentane Schleifendurchlauf abgebrochen, dann aber mit der nächsten Iteration fortgesetzt. `break` und `continue` sind also analog zu `exit` und `cycle` von F. Allerdings kann man bei `break` und `continue` in C keine Labels angeben.

Logische Verzweigungen

Die bedingte Ausführung bestimmter Programmblöcke erreicht man in beiden Sprachen durch `if`-Konstruktionen. Die allgemeine Form in F ist

```
if(condition_1) then
  block_1
  ...
else if(condition_2) then
  block_2
  ...
...
else
  else_block
  ...
end if
```

Hier werden der Reihe nach die Bedingungen *condition_1*, *condition_2* usw. geprüft und der erste Block von Befehlen (und nur dieser) ausgeführt, für den die Bedingung erfüllt ist. Trifft keine der Bedingungen zu, wird der `else`-Block ausgeführt. Eine `if`-Konstruktion kann auch nur aus den `if`...`else`...`end if`-Blöcken bzw. einem einzigen `if`...`end if`-Block bestehen.

In C funktioniert die `if`-Konstruktion genauso wie in F und hat die Gestalt

```
if(condition_1) {
  block_1
  ...
} else if(condition_2) {
  block_2
  ...
...
} else {
  else_block
  ...
}
```

Auch hier dürfen die `else if`- und `else`-Zweige fehlen.

Die Bedingungen müssen in F logische Ausdrücke sein; in C können es sogar Ausdrücke allgemeiner Art sein. In der Praxis sind es jedoch in beiden Fällen fast immer numerische Vergleiche oder Tests der Gleichheit von Zeichen (in F auch von Zeichenketten). In F stehen beim Bilden logische Ausdrücke außerdem noch Variablen vom Typ `logical` zur Verfügung.

Die numerischen Vergleichsoperatoren sind in F und C fast identisch

F	C	Bedeutung
`<`	`<`	kleiner
`>`	`>`	größer
`<=`	`<=`	kleiner gleich
`>=`	`>=`	größer gleich
`==`	`==`	gleich
`/=`	`!=`	ungleich

Logische Ausdrücke können mit Hilfe der folgenden Operatoren verknüpft bzw. negiert werden

F	C	Bedeutung
`.or.`	`||`	oder
`.and.`	`&&`	und
`.not.`	`!`	nicht

(Auch hier sind in F die Punkte wieder Bestandteil des Operators.)

Arithmetische Ausdrücke und eingebaute Funktionen

Sowohl F als auch C verwenden für die vier Grundrechnungsarten die üblichen Symbole, nämlich die Operatoren

`+`, `-`, `*` und `/`

In F gibt es zusätzlich noch den Potenzierungsoperator `"**"`, d.h. für x^y schreibt man

```
x**y
```

wobei der Exponent auch nicht-ganzzahlig sein darf (also wäre etwa `x**0.5` die Quadratwurzel aus `x`).

Arithmetische Ausdrücke werden in der gewohnten Weise ausgewertet: Potenzierung (so vorhanden) hat Vorrang vor Multiplikation und Division, die ihrerseits Vorrang vor Addition und Subtraktion haben. Will man eine andere Reihenfolge der Auswertung erreichen, so kann man das durch Klammerung von Teilausdrücken mit runden Klammern `(...)` erzwingen.

Zusätzlich stehen in beiden Sprachen eine ganze Reihe von "eingebauten" mathematischen Funktionen mit z.T. identischen Namen zur Verfügung. Die wichtigsten sind

F	C	Anmerkungen
`abs(x)`	`abs(i)`, `fabs(x)`	
`acos(x)`	`acos(x)`	
	`acosh(x)`	
`aimag(z)`		Imaginärteil
`asin(x)`	`asin(x)`	
	`asinh(x)`	
`atan(x)`	`atan(x)`	
`atan2(y,x)`	`atan2(y,x)`	Hauptwert des Arguments von $x + iy$
`cmplx(x,y)`		Konversion in komplexe Zahl $x + iy$
`conjg(z)`		komplexe Konjugation
`cos(x)`	`cos(x)`	
`cosh(x)`	`cosh(x)`	
	`erf(x)`, `erfc(x)`	(komplementäre) Errorfunktion
`exp(x)`	`exp(x)`	
`int(x)`		Konversion in ganze Zahl (mit Abrunden nach 0)
	`j0(x)`, `j1(x)`, `jn(n,x)`	Besselfunktion 1. Art
`log(x)`, `log10(x)`	`log(x)`, `log10(x)`	Logarithmus zur Basis e bzw. 10
`max(x1,x2,...)`		Maximum von x_1, x_2,...
`min(x1,x2,...)`		Minimum von x_1, x_2,...
`modulo(x,y)`	`fmod(x,y)`	x mod y
`nint(x)`		Konversion in nächste ganze Zahl
	`pow(x,y)`	x^y
`real(x)`		Konversion in Typ `real`
`sin(x)`	`sin(x)`	
`sinh(x)`	`sinh(x)`	
`sqrt(x)`	`sqrt(x)`	
`tan(x)`	`tan(x)`	
`tanh(x)`	`tanh(x)`	
	`y0(x)`, `y1(x)`, `yn(n,x)`	Besselfunktion 2. Art

Wenn nicht anders angegeben, sind für die mathematischen Funktionen in C sowohl Argument(e) als auch Resultat vom Typ `double`. Ausnahmen bilden die Funktionen `abs` (Absolutbetrag einer ganzen Zahl) sowie `jn` und `yn`, die natürlich ein ganzzahliges Argument verlangen. Normalerweise kann man statt der `double`-Argumente auch solche vom Typ `float` verwenden.

In F sind die meisten eingebauten Funktionen "generisch", d.h. wenn immer das sinnvoll ist, können die Argumente von beliebigem Typ sein; der Typ des Ergebnisses richtet sich dann nach dem Typ des Arguments. Das bezieht sich sowohl auf den numerischen Typ und die Präzision als auch darauf, ob es sich um primitive Datentypen oder höherdimensionale Objekte (Vektoren, Matrizen usw.) handelt.

Funktionen

Funktionen und Unterprogramme

In F unterscheidet man bei Unterprogrammen (Prozeduren) zwischen "Funktionen" und "Unterprogrammen" im eigentlichen Sinn. Eine Funktion ist eine Prozedur, an die Parameter übergeben werden können und die einen Wert (oder, z.B. im Fall einer vektorwertigen Funktion, mehrere Werte) zurückliefert, der unter dem Namen der Funktion in Ausdrücken verwendet werden kann, wie das etwa bei den "eingebauten" mathematischen Funktionen der Fall ist. Damit Funktionen frei von Nebeneffekten sind, dürfen die übergebenen Argumente innerhalb der Funktion nicht verändert werden.

Ein Unterprogramm (s. später) in F ist ebenfalls eine Prozedur, die aber keinen "Funktionswert" zurückliefert, sondern es wird bei den Parametern zwischen veränderbaren und nicht veränderbaren unterschieden. Nur die ersteren dürfen vom Unterprogramm gesetzt werden, und sie werden dazu benützt, die "Resultate" der Prozedur an das rufende Programm zu übertragen. (Außerdem haben Unterprogramm auch noch Schreibzugriff auf globale Variablen.)

Im Gegensatz dazu kennt C *nur* Funktionen, d.h. jede Prozedur, sofern sie nicht vom Typ `void` ist, kann einen Wert zurückliefern. Dies kann ein Funktionswert im engeren Sinn oder aber auch nur ein Fehlercode sein.

Call by name und call by value

F und C unterscheiden sich grundlegend im Mechanismus, wie Argumente an Prozeduren übertragen werden. Während in F die Speicheradressen der Parameter an die Prozedur übergeben werden ("call by name") und ein Unterprogramm somit direkten Zugriff auf gewisse Variable des rufenden Programms hat und diese u.U. auch verändern kann, werden in C nur die Werte der Parameter übertragen ("call by value"). Werden sie von der Prozedur verändert, so wirkt sich das auf die Variablen des rufenden Programms nicht aus.

Sollen also in C tatsächlich die Variablen des rufenden Programms verändert werden, so muß man statt der Werte explizit ihre Adressen an die Prozedur übergeben. Dies geschieht durch Voranstellen des `"&"`-Zeichens vor den Variablennamen in der Parameterliste des Prozeduraufrufs. (In der Prozedur selbst müssen diese Adressen dann als Pointer deklariert und behandelt werden.)

Syntax

Struktur und Syntax einer Funktion in F sehen folgendermaßen aus

```
function function_name(arg_1,arg_2,...) result(result)
  type_1,intent(in)::arg_1
  type_2,intent(in)::arg_2
  ...
  type::result
  ...
  result=...
  ...
  return
end function function_name
```

function_name ist der Name der Funktion, *arg_1*, *arg_*,... die Parameterliste und *result* der Name einer lokalen Variable, der irgendwo in der Prozedur der Funktionswert zugewiesen wird. Die Parameter müssen durch das Attribut `intent(in)` als in der Prozedur unveränderlich gekennzeichnet werden. Ein analoges `intent(out)` bei der Deklaration von *result* kann entfallen, da es sich von selbst versteht, daß es möglich sein muß, den Funktionswert zu setzen (überschreiben).

Dieselbe Konstruktion hat in C die Gestalt

```
type function_name(type_1 arg_1,type_2 arg_2,...) {
  ...
  return expression;
}
```

Der Typ des Funktionswertes kann hier gleich vor dem Namen der Funktion angegeben werden. Ebenso können die Parameter bereits in der Parameterliste deklariert werden. Der im Ausdruck *expression* berechnete eigentliche Funktionswert wird durch den `return`-Befehl an das rufende Programm zurückgegeben. Es ist also in C nicht notwendig, für den Funktionswert extra eine Variable anzulegen.

Module und Header-Files

Das Konzept des Moduls in F bietet die Möglichkeit, Deklarationen und Definitionen von Typen, Variablen, Operatoren und Prozeduren in Gruppen zusammenzufassen und innerhalb eines größeren Programms global oder auch nur selektiv lokal "sichtbar" zu machen. Verwendet ein F-Programm selbst geschriebene Funktionen oder Unterprogramme, so müssen sie auf jeden Fall in einem Modul definiert und mit einer `use`-Direktive dem rufenden Programm bekannt gemacht werden.

Die Struktur eines Moduls ist

```
module module_name
  Deklarationen und Definitionen
  ...
[contains
  Funktionen und Unterprogramme
  ...]
end module module_name
```

Da alle Variablen, Funktionsnamen usw. eines Moduls außerhalb desselben zunächst unbekannt sind, muß man alles, was für das rufende Programm sichtbar sein soll, erst explizit mit Hilfe einer `public`-Deklaration "sichtbar" machen.

Will man also beispielsweise eine Funktion $f(x)$ verwenden, dann müßte das entsprechende Modul mindestens folgende Definitionen enthalten

```
module module_name
  ...
  public::f
  ...
contains
  ...
  function f(x) ...
    ...
  end function f
  ...
end module module_name
```

Außerdem müßte im rufenden Programm (bzw. in dem Modul, in dem eine rufende Prozedur steht) dieses Modul im Deklarationsteil mit einer `use`-Direktive

```
use module_name
```

eingebunden werden

Die sogenannten "eingebauten" Funktionen und Prozeduren von F sind Bestandteil des Sprachstandards. Der Compiler hat daher von vornherein die notwendigen Informationen, um in der Compilationsphase zu prüfen, ob z.B. Funktionen mit der richtigen Anzahl und dem korrekten Typ von Argumenten aufgerufen werden. Ebenso werden beim Linken diese Prozeduren automatisch in den Systembibliotheken gesucht. Um das Einbinden von Modulen, die Angabe von Bibliotheken beim Linken usw. muß man sich also nur bei selbst geschriebenen Prozeduren kümmern.

Im Gegensatz dazu sind in C auch grundlegende Funktionen wie Ein- und Ausgabe oder die mathematischen Funktionen *nicht* Bestandteil der Sprache, sondern nur in Form von Bibliotheken vorhanden, die von Implementation zu Implementation leicht variieren können. Damit der Compiler prüfen kann, ob z.B. Funktionsaufrufe formal korrekt durchgeführt werden, muß man ihm zusätzliche Informationen zur Verfügung stellen. Diese Informationen stehen in den "Header-Files", die typischerweise am Beginn des Programms mit Hilfe von `include`-Direktiven vom Preprocessor eingefügt werden.

Will man also z.B. sowohl die Ein- und Ausgabe- als auch die mathematischen Funktionen von C verwenden, so müßten mit

```
#include <stdio.h>
#include <math.h>
```

die Header-Files `stdio.h` und `math.h` eingefügt werden.

Zusätzlich kann es noch notwendig sein, beim Linken die entsprechenden Bibliotheken anzugeben. Welche Header-Files und welche Bibliotheken man angeben muß, kann man u.U. der `man`-Page der verwendeten Funktion entnehmen.