Programmieren für Mathematiker WS2017/18

Dozent: Prof. Dr. Wolfgang Walter

15. August 2018

Inhaltsverzeichnis

Kapitel I

allgemeine Informationen

Eine Programmiersprache ist lexikalisch, syntaktisch und semantisch eindeutig definiert. Eine Compiler übersetzt die Programmiersprache in Maschinensprache . Ein Interpreter arbeitet das Programm dann ab. Ein Laufzeitsystem stellt grundlegende Operationen und Funktionen zur Verfügung.

1. Bereiche der Informatik

Die Informatik untergliedert sich in 4 Bereiche:

- Technische Informatik
- Praktische Informatik
- Theoretische Informatik
- Angewandte Informatik

Die Technische Informatik beschäftigt sich mit der Konstruktion der Hardware, zum Beispiel der Datenleitungen, um Informationen durch das Internet zu transportieren. Wichtige Firmen sind hier: Intel, Globalfoundries und Infineon.

Die Praktische Informatik beschäftigt sich mit der Software, also Betriebssystem, Compiler, Interpreter und so weiter. In alltäglicher Software findet sich rund 1 Fehler in 100 Zeilen Quelltext. In wichtiger Software, also Raketen, Betriebssysteme, ..., ist es nur 1 Fehler pro 10.000 Zeilen Code.

Die Theoretische Informatik beschäftigt sich mit Logik, formalen Sprachen, der Automatentheorie, Komplexität von Algorithmen, ...

Die Angewandte Informatik beschäftigt sich mit der Praxis, dem Nutzer, der Interaktion zwischen Mensch und Maschine, ...

2. Maßeinheiten und Größenordnungen

Ein bit ist ein Kunstwort aus “binary“ und “digit“. Es kann nur 2 Werte speichern: 0 und 1

Ein nibble ist eine Hexadezimalziffer, bündelt also 4 bits und kann damit 16 Werte annehmen: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E und F.

Ein byte bündelt 2 nibble, also 8 bit. Er ist die gebräuchlichste, direkt addressierbare, kleinste Speichereinheit. Weitere Speichergrößen sind:

Name	**Anzahl byte**	**Name**	**Anzahl byte**
1 KB	10^3	1 KiB	$2^{10} = 1.024$
1 MB	10^6	1 MiB	$2^{20} = 1.048.576$
1 GB	10^9	1 GiB	$2^{30} = 1.073.741.824$
1 TB	10^{12}	1 TiB	2^{40}
1 PB	10^{15}	1 PiB	2^{50}
1 EB	10^{18}	1 EiB	2^{60}

Der ROM (“read-only-memory“) speichert wichtige Informationen auch ohne Strom, wie zum Beispiel die Uhrzeit, Informationen über die Festplatte, ... Er ist nicht mehr änderbar, außer durch Belichtung.

Der RAM (“random-access-memory“) ermöglicht den Zugriff auf alle Adressen, insbesondere im Hauptspeicher.

Kapitel II

Zahldarstellungen

1. Basis-Konvertierung ganzer Zahlen

Die Notation $[9]_{10}$ bedeutet, dass man die Zahl 9 im Zehner-System betrachtet. Es gilt also $[9]_{10} = [1001]_2$ und $[10]_{10} = [1010]_2$.

Um eine Zahl von einer gegebenen Basis in eine Zielbasis b zu konvertieren, so teilt man immer wieder durch b und notiert den Rest als nächste Ziffer von hinten nach vorne. Am Beispiel von $[57]_{10}$ ins Zweier-System sieht das so aus:

$$
\begin{aligned}
\frac{57}{2} &= 28 \text{ Rest } 1 \Rightarrow \text{ letzte Ziffer der Binärdarstellung}\\
\frac{28}{2} &= 14 \text{ Rest } 0 \Rightarrow \text{ vorletzte Ziffer der Binärdarstellung}\\
\frac{14}{2} &= 7 \text{ Rest } 0\\
\frac{7}{2} &= 3 \text{ Rest } 1\\
\frac{3}{2} &= 1 \text{ Rest } 1\\
\frac{1}{2} &= 0 \text{ Rest } 1
\end{aligned}
$$

Also gilt: $[57]_{10} = [111001]_2$.

Die umgekehrte Richtung verläuft ähnlich:

$$
\begin{array}{rcccl}
111001 & : & 1010 & = & 101 \;\text{R}\; 111\\
\text{-}1010 & & & & \\
\hline
01000 & & & & \\
\text{-}00000 & & & & \\
\hline
10001 & & & & \\
\text{-}01010 & & & & \\
\hline
111 & & & &
\end{array}
$$

Also $[111001]_2$ durch $[10]_{10} = [1010]_2$ gleich $[101 \text{ Rest } 111]_2 = [5 \text{ Rest } 7]_{10} \Rightarrow [57]_{10}$.

Von Basis 2 in Basis 4, 8 oder 16 ist dann ganz einfach: $[111100101]_2$

- Zweiergruppen von hinten nach vorne zusammenzählen: $[13211]_4$
- Dreiergruppen von hinten nach vorne zusammenzählen: $[745]_8$
- Vierergruppen von hinten nach vorne zusammenzählen: $[1E5]_{16}$

2. Basis-Konversion gebrochener Zahlen

Festkommadarstellung (nur Betrag der Zahl, ohne Vorzeichen):

Gewichte	B^k	B^{k-1}	...	B^1	B^0	.	B^{-1}	B^{-2}	...	B^{-l}
Ziffern	m_k	m_{k+1}	...	m_{-1}	m_0	.	m_1	m_2	...	m_l

Also: $\sum\limits_{i=k}^{l} m_i \cdot B^{-i}$.

Die Konvertierung des ganzzahligen Anteils vor dem "." läuft wie gehabt. Um den gebrochenen Anteil zu konvertieren, multipliziert man wiederholt mit der Zielbasis b und nimmt den jeweiligen ganzzahligen Anteil als Nachkommaziffern (von links nach rechts). Mit dem gebrochenen Anteil macht man weiter. Wir wollen die Zahl $[0.625]_{10}$ ins Zweiersystem konvertieren:

$$\begin{aligned} 0.625 \cdot 2 &= \mathbf{1}.25 \\ 0.25 \cdot 2 &= \mathbf{0}.5 \\ 0.5 \cdot 2 &= \mathbf{1} \end{aligned}$$

Also gilt: $[0.625]_{10} = [0.101]_2$.

Wieder anders herum:

$$\begin{aligned} 0.101 \cdot 1010 &= \mathbf{110}.010 \\ 0.010 \cdot 1010 &= \mathbf{10}.100 \\ 0.100 \cdot 1010 &= \mathbf{101}.0 \end{aligned}$$

Also gilt $[0.101]_2 = [0.110|10|101]_2 = [0.625]_{10}$.

Jetzt wollen wir $[0.1]_{10}$ ins Zweiersystem konvertieren:

$$\begin{aligned} 0.1 \cdot 2 &= \mathbf{0}.2 \\ 0.2 \cdot 2 &= \mathbf{0}.4 \quad (1) \\ 0.4 \cdot 2 &= \mathbf{0}.8 \\ 0.8 \cdot 2 &= \mathbf{1}.6 \\ 0.6 \cdot 2 &= \mathbf{1}.2 \\ 0.2 \cdot 2 &= \mathbf{0}.4 \quad (2) \end{aligned}$$

Wie man sieht, sind die Zeilen (1) und (2) gleich, das heißt, diese Konvertierung wird unendlich lange laufen. Also: $[0.1]_{10} = [0.0\overline{0011}]_2$. Aber es muss gelten: $[0.1]_{10} \cdot [10]_{10} = [1]_{10}$. Aber es stimmt: $[0.0\overline{0011}]_2 \cdot [1010]_2 = [0.\overline{1}]_2 = [1]_2$.

Entsprechend gilt:

$$
\begin{aligned}
[0.2]_{10} &= [0.\overline{0011}]_2 \\
[0.3]_{10} &= [0.01\overline{0011}]_2 \\
[0.4]_{10} &= [0.011\overline{0011}]_2 \\
[0.5]_{10} &= [0.1]_2 \\
[0.6]_{10} &= [0.1\overline{0011}]_2 \\
[0.7]_{10} &= [0.1011\overline{0011}]_2 \\
[0.8]_{10} &= [0.11\overline{0011}]_2 \\
[0.9]_{10} &= [0.111\overline{0011}]_2
\end{aligned}
$$

Problem: Rundungen schon bei $\frac{1}{10}$ $\Rightarrow$ falsche Nachkommastellen. Die Lösung sind hier Gleitkommazahlen.

3. Gleitkommazahlen

Gleitkommazahlen werden auch Fließkommazahlen, Gleitpunktzahlen, Fließpunktzahlen oder floating-point-numbers genannt.

Das Gleitkommaformat $R = (b, l, \underline{e}, \overline{e})$ besteht aus

- einer Basis b
- einer Mantissenlänge l
- einem Exponentenbereich von $\underline{e}$ bis $\overline{e}$.

Eine Gleitkommazahl ist entweder 0 oder $x = (-1)^s \cdot m \cdot b^e$ mit

- Vorzeichenbit $s \in \{0, 1\}$
- Mantisse $m = [0.m_1 m_2 m_3 ... m_l]_b$ mit Mantissenziffern $m_i \in \{0, 1, 2, ..., b-1\}$
- $e \in \{\underline{e}, \underline{e}+1, \underline{e}+2, ..., \overline{e}\}$

Schauen wir uns das Beispiel $R(2, 3, -1, +2)$ an. Eine solche Zahl benötigt 1 bit für s, 2 bits für e und 3 bits für m.

$m =$ **0.**	**111**	**110**	**101**	**100**	**011**	**010**	**001**	**000**
$e = -1$	$\frac{7}{16}$	$\frac{6}{16}$	$\frac{5}{16}$	$\frac{4}{16}$	$\frac{3}{16}$	$\frac{2}{16}$	$\frac{1}{16}$	0
$e = 0$	$\frac{14}{16}$	$\frac{12}{16}$	$\frac{10}{16}$	$\frac{8}{16}$	$\frac{6}{16}$	$\frac{4}{16}$	$\frac{2}{16}$	0
$e = 1$	$\frac{28}{16}$	$\frac{24}{16}$	$\frac{20}{16}$	$\frac{16}{16}$	$\frac{12}{16}$	$\frac{8}{16}$	$\frac{4}{16}$	0
$e = 2$	$\frac{56}{16}$	$\frac{48}{16}$	$\frac{40}{16}$	$\frac{32}{16}$	$\frac{24}{16}$	$\frac{16}{16}$	$\frac{8}{16}$	0

Es gibt also auch mehrere Darstellungen für eine Zahl! Die Cyan eingefärbten Zahlen können auch anders dargestellt werden.

Grüne Zahlen sind sogenannte normalisierte Gleitkommazahlen , ihre erste Mantissenziffer ist $\neq 0$. Die roten Zahlen sind denormalisierte Gleitkommazahlen : Ihre ersten Mantissenziffer ist $m_1 = 0$ und ihr Exponent $e = \underline{e}$. Da das erste Mantissenbit häufig eine 1 ist, wird angenommen, dass das erste Mantissenbit eine 1 ist und wird deswegen nicht gespeichert (hidden bit). Das sorgt dafür, dass bei 3 bit Genauigkeit mit 4 bit Genauigkeit gerechnet werden kann. Ist das erste Mantissenbit eine 0, gibt es dafür eine spezielle Exponentenkennung.

Ein Zahlenstrahl mit diesen Zahlen ist besonders dicht um 0, aber ab 2 werden die Abstände sehr groß.

Die größte darstellbare Zahl ist $x_{max} = 0.1111...1 = (1 - b^l) \cdot b^{\overline{e}}$.
Der kleinste darstellbare normalisierte Betrag ist $x_{min,N} = 0.10000...0 = b^{\underline{e}-1}$.
Der kleinste darstellbare denormalisierte Betrag ist $x_{min,D} = 0.0000...1 = b^{\underline{e}-l}$.

Doch es gibt eine Probleme:

- absolute/relative Fehler bei Zahlen, die zwischen 2 darstellbaren Zahlen liegen $\Rightarrow$ Rundungen bei nahezu jeder Rechnung!
- Grundrechenarten können nicht darstellbare Zahlen erzeugen

4. Rundung

Eine Rundung ist eine Funktion $O : \mathbb{R} \to$ Gleitkomma-Raster R.

Eine Rundung O hat folgende Eigenschaften:

1. $O(x) = x$ wenn $x \in R$
2. $x, y \in \mathbb{R}$ mit $x < y \Rightarrow O(x) < O(y)$
3. $O(-x) = -O(x)$, nur manche Rundungen haben diese Eigenschaft

Es gibt verschiedene Rundungsmodi:

- "to nearest": zur nächsten Gleitkommazahl, wenn 2 Gleitkommazahlen gleich weit weg sind, wird abwechselnd auf- und abgerundet
- "trancation": Abschneiden der Nachkommastellen $\Rightarrow$ betragskleiner runden
- "augmentation": zusätzliche Stellen hinzufügen $\Rightarrow$ betragsgrößer runden
- "upward": nach oben runden
- "downward": nac unten runden

Wenn O eine Rundung mit einem Rundungsmodus, also $O \in \{\text{Rundungsmodi}\}$, ist und $\circ$ eine Grundrechenart, also $\circ \in \{+, -, \cdot, \div\}$, dann gilt für eine Gleitkommaoperation $\odot$

$$x, y \in R : x \odot y := O(x \circ y)$$

Auslöschung in Summen von Gleitkommazahlen tritt auf, wenn die Größenordnung der exakten Summe wesentlich kleiner ist als die Größenordnung der Summanden (bzw. der Zwischenergebnisse).

Kapitel III

Grundstrukturen von Algorithmen

1. Begriffe

Eine Sequenz sind einzelne Anweisungen hintereinander.
Eine Selektion ist eine Verzweigung.
Eine Repetition ist eine Wiederholung.

1.1. Variablen und Daten

Wir sehen uns einen Biertrinker an, der nach dem Genuss noch ein paar Besorgungen machen muss. Dabei ergeben sich folgende Variablen

- Variable **Durst** von Typ *LOGICAL*
- Variable **Geld** von Typ *INTEGER*
- Variable **PreisDerBesorgung** von Typ *INTEGER*
- Variable **Rest** von Typ *INTEGER*
- Variable **Bierpreis** von Typ *INTEGER*
- Variable **WirtschaftAnnehmbar** von Typ *LOGICAL*
- Variable **Autofahrer** von Typ *LOGICAL*
- Variable **AlkoholGrenzwert** von Typ *REAL*
- Variable **AlkoholVergiftungsWert** von Typ *REAL*

1.2. Schleifen

In Fortran gibt es 4 Arten von Schleifen :

- Endlosschleife
- Schleife mit Anfangsbedingung
- Schleife mit Endbedingung
- Zählschleife

Bei einer Endlosschleife wird der Anweisungsblock innerhalb der Schleife unendlich lange ausgeführt:

```
do
 Anweisung1
 Anweisung2
 Anweisung3
```

```
end do
```

Bei einer Schleife mit Anfangsbedingung wird der Anweisungsblock nur ausgeführt, wenn die Anfangsbedingung wahr ist. Ist sie wahr, so wird der Block ausgeführt und anschließend überprüft, ob die Anfangsbedingung wieder wahr ist. Ist die Anfangsbedingung nicht wahr, so wird die Schleife nicht ausgeführt.

```
do while(Anfangsbedingung)
 Anweisung1
 Anweisung2
 Anweisung3
end do
```

Hat die Schleife eine Endbedingung , so wird der Anweisungsblock auf jedem Fall 1-mal ausgeführt. Erst dann wird überprüft, ob die Endbedingung wahr ist. Ist sie das, wird die Schleife **verlassen**. Ist sie falsch, so wird die Schleife erneut ausgeführt.

```
do
 Anweisung1
 Anweisung2
 Anweisung3
 if(Endbedingung) exit
end do
```

Eine Zählschleife in Fortran ist ähnlich wie in anderen Programmiersprachen konzipiert, aber die Syntax ist deutlich verschieden. Eine Zählschleife besitzt eine Zählvariable (die auch in der Schleife benutzt werden kann, aber nicht geändert werden sollte), die von einer Anfangszahl mit bestimmter Schrittweite solange hochgezählt (oder bei negativer Schrittweite heruntergezählt) wird, bis die Zählvariable die Endzahl erreicht.

```
do i = anfang, ende, schrittweite
 Anweisung1
 Anweisung2
 Anweisung3
end do
```

Bitte beachten:

- Der Zustand der Zählvariable `i` vor der Schleife geht verloren, auch wenn die Schleife 0-mal läuft.
- Die Zählvariable `i` darf im Inneren der Schleife nicht geändert werden.
- Der Endzustand der Zählvariable `i` ist nach der Schleife nicht definiert.
- Ausdrücke werden zu Beginn genau 1-mal (vor der ersten Iteration) berechnet und sind dann fest.
- Die Anzahl der Iterationen ist: $N = \max\left\{0, \mathtt{nint}\left(\frac{\mathtt{ende\ -\ anfang\ +\ i}}{\mathtt{i}}\right)\right\}$

2. Einfache Syntax

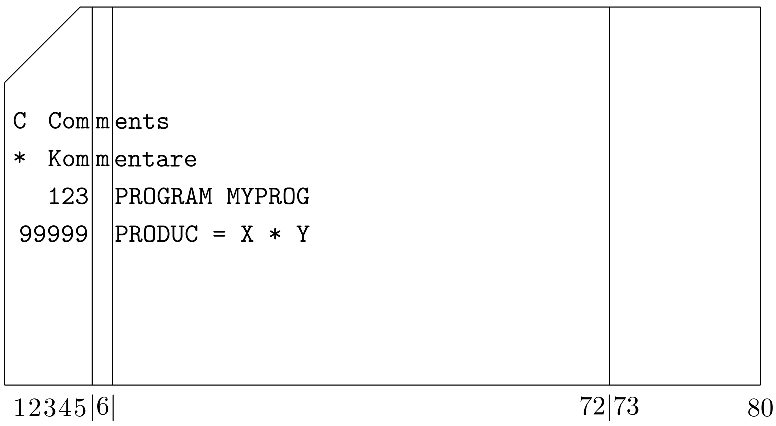
Es gibt einen zulässigen Fortran-Zeichensatz. Dieser umfasst zum Beispiel die Buchstaben A-Z und a-z, sowie 0-9, Sonderzeichen und Operatoren.

Nun einige lexikalische Einheiten (Symbole/Tokens):

- Keywords sind nicht reserviert, Variablen können also auch nach Keywords benannt werden.
- Identifiers (Namen) haben eine Länge von maximal 63 Zeichen. Variablen können Buchstaben, Zahlen und auch den Unterstrich enthalten.
- Literale (Konstanten): 3 ⇒ INTEGER, 2.876 ⇒ REAL, .TRUE. ⇒ LOGICAL, "Hallo" ⇒ STRING
- Labels (Marken): 00000 ... 99999 sind Sprungmarken, die mit `GOTO 99999` erreicht werden können.
- Separatoren (Trennsymbole) sind: `()`, `/`, `/(/)`, `[]`, `=`, `=>`, `:`, `::`, `,`, `;` und `%`.
- Operatoren sind: `+`, `-`, `*`,`/`, `**`, `//`, `==`, `<=`, `<`, `/=`, `>`, `>=`, `.NOT.`, `.OR.`, `.AND.`, `.EQV.` und `.NEQV.`.

In den alten Quellformen bis vor Fortran-90, also insbesondere Fortran-66 und Fortran-77 waren

- Namen maximal 6 Zeichen lang und
- Lochkarten 80 Zeichen breit.



Ab Fortran-90 wurden neue Quellformen entwickelt, das heißt:

- Zeilen sind nun maximal 132 Zeichen lang
- Kommentare beginnen mit "!" und gehen bis zum Zeilenende
- Eine neue Zeile ist eine neue Anweisung, außer die letzte Zeile endet mit "&"
- "&" am Zeilenende bedeutet, dass die nächste nicht-Kommentar und nicht-Leerezeile die Anweisung fortsetzt.
- Die Fortsetzung darf mit "&" beginnen.
- Es sind maximal 39 Fortsaetzungszeilen möglich
- Leerzeichen sind signifkant ⇒ alle lexikalischen Tokens sind am Stück zu schreiben.
- Groß- und Kleinschreibung ist nicht signifikant in Namen und Keywords

3. Datentypen

Fortran besitzt 5 Datentypen. Für jeden Datentyp gibt es spezielle dazugehörige Funktionen:

- **Integer** für ganze Zahlen
- **Real** für reelle Zahlen
- **Complex** für komplexe Zahlen
- **Logical** für logische Werte
- **Character** für Strings

3.1. Der Datentyp `Integer`

mögliche Argumente	`integer(kind=...)`
Darstellung	$\{\langle Z \rangle\}$
Wert	mindestens 1 Ziffer, höchstens unendlich
Wertemenge	normalerweise im 2er-Komplement: $\left[-2^{l-1}, 2^{l-1}-1\right]$
Operationen	`+`, `-`, `*`, `/` (schneidet Nachkommastellen ab), `**` (schneidet Nachkommastellen ab)

Wichtige Funktionen:

- `sign(x,y)` gibt den Betrag von x, wenn $y \geq 0$ und $-|x|$, wenn $y < 0$
- `int(x)` schneidet die Nachkommastellen von x ab
- `floor(x)` rundet ab ($\lfloor x \rfloor$)
- `ceiling(x)` rundet auf ($\lceil x \rceil$)
- `selected_int_kind(k)` liefert KIND-Parameter des kleinsten INTEGER-Typs, der dem alle Zahlen mit k Stellen darstellen kann

3.2. Der Datentyp `Real`

mögliche Argumente	`real(kind=...)`
Darstellung	$\{\langle Z \rangle\}.\{\langle Z \rangle\}E \pm \{\langle Z \rangle\}$
Wert	mindestens 1 Ziffer, höchstens unendlich
Wertemenge	
Operationen	`+`, `-`, `*`, `/`, `**`

Wichtige Funktionen:

- `aint(x)` schneidet die Nachkommastellen von x ab
- `real(x)` konvertiert zu REAL

- `selected_real_kind(p,r)` liefert KIND-Parameter mit p Ziffern in der Mantisse und r Ziffern im Exponenten

3.3. Der Datentyp `Complex`

mögliche Argumente	`complex(kind=...)`
Darstellung	$(\langle\mathfrak{Re}\rangle, \langle\mathfrak{Im}\rangle)$
Wert	$\mathfrak{Re}$,$\mathfrak{Im}$ vorzeichenbehaftete realle Konstanten
Wertemenge	
Operationen	`+`, `-`, `*`, `/`, `**`

Wichtige Funktionen:

- `abs(c)` liefert den Betrag von c, also $\sqrt{x^2+y^2}$
- `real(c)` liefert den Realteil von c
- `aimag(c)` liefert den Imaginärteil von c
- `conj(c)` liefert das konjugiert Komplexe zu c

3.4. Der Datentyp `Logical`

mögliche Argumente	
Darstellung	`.TRUE.`, `.FALSE.`
Wert	`.TRUE.` oder `.FALSE.`
Wertemenge	`.TRUE.,.FALSE.`
Operationen	`.AND.`, `.OR.`, `.NOT.`, `.EQV.`, `.NEQV`

3.5. Der Datentyp `Character`

mögliche Argumente	`character(len=...)`
Darstellung	Zeichen
Wert	einzelnes Zeichen oder Zeichenketten
Wertemenge	alle möglichen Zeichenfolgen mit l Zeichen
Operationen	`//` (Konkardination: fügt 2 Strings zusammen)

Wichtige Funktionen:

- `ichar(c)` gibt den internen ganzzahligen Zeichencode von c
- `char(i)` gibt den Zeichencode zu c
- ⟨Zeichenkette⟩`(a:b)` gibt den Teilstring vom a-ten bis zum b-ten Zeichen

- `len(zk)` gibt die Länge der Zeichenkette *zk*
- `trim(zk)` liefert die Zeichenkette ohne anhängende Leerzeichen
- `adjustl(zk)` Inhalt der Zeichenkette wird nach vorne geschoben
- `repeat(zk,copies)` gibt einen String mit *copies*-facher Zeichenkette
- `index, scan, verify` durchsucht einen String

Es gibt allerdings noch eine Reihe weiterer (mathematischer) Funktionen, das Ergebnis ist selbsterklärend:

- `sin(x)`, `asin(x)`, `sinh(x)`
- `cos(x)`, `acos(x)`, `cosh(x)`
- `tan(x)`, `atan(x)`, `atan2(x,y)=atan(x/y)`
- `sqrt(x)`
- `exp(x)`
- `log10(x)`

3.6. INTEGER-Division

Die klassiche INTEGER-Division sieht so aus:

Division:	$\frac{a}{b} =$ `a/b`
Rest der Division:	$a - \left(\frac{a}{b}\right) \cdot b =$ `mod(a,b)`
Beispiele (Division)	**Beispiele (Rest)**
$\frac{8}{5} \to 1$	`mod(8,5)` $\to$ 3
$\frac{-8}{5} \to -1$	`mod(-8,5)` $\to$ -3
$\frac{-8}{-5} \to 1$	`mod(-8,-5)` $\to$ -3
$\frac{8}{-5} \to -1$	`mod(8,-5)` $\to$ 3

Das darf man aber nicht mit der nach unten abgerundeten REAL-Division verwechseln:

Division:	$\left\lfloor\frac{a}{b}\right\rfloor =$ `floor(a/b)` $=$ `floor(real(a)/real(b))`
Rest der Division:	$a - \left\lfloor\frac{a}{b}\right\rfloor \cdot b =$ `modulo(a,b)`
Beispiele (Division)	**Beispiele (Rest)**
$\frac{8.0}{5.0} \to 1$	`modulo(8,5)` $\to$ 3
$\frac{-8.0}{5.0} \to -2$	`modulo(-8,5)` $\to$ 2
$\frac{-8.0}{-5.0} \to 1$	`modulo(-8,-5)` $\to$ -3
$\frac{8.0}{-5.0} \to -2$	`modulo(8,-5)` $\to$ -2

3.7. Potenzieren

Die Potenz-Operation `**` ist die einzige Operation die von rechts nach links gelesen wird, bei allen anderen Operationen wird in Leserichtung, also von links nach rechts gearbeitet.

- `2**3` $= 2^3 = 8$
- `2**(-3)` $\rightarrow$ `int(`2^{-3}`)=int(`$\frac{1}{8}$`)` $= 0$
- `(-3)**2` $= (-3)^2 = 9$
- `-3**2` $= -3^2 = -9$
- `2**3**2` $=$ `2**(3**2)` $= 2^9 = 512$
- `(2**3)**2` $= (2^3)^2 = 64$

3.8. Operator-Prioritäten

Operatoren haben in Fortran eine Priorität, die weiter gefasst ist als: “Punktrechnung vor Strichrechnung“. Operatoren mit der höchsten Priorität (12) werden zuerst ausgeführt; Operatoren mit der Priorität 1 zuletzt.

1. selbstdefinierte Operatoren binär
2. `.EQV.` und `.NEQV.`
3. `.OR.`
4. `.AND.`
5. `.NOT.`
6. Vergleichsoperatoren
7. `//`
8. `+`, `-` als Addition beziehungsweise Subtraktion
9. `+`, `-` als Vorzeichen
10. `*`, `/`
11. `**`
12. selbstdefinierte Operatoren unär

Jetzt noch ein kurzer Abschnitt zu Variablen in imperativen Programmiersprachen. Eine Variable wird durch ein 5-Tupel (N,T,G,L,R) beschrieben, das heißt

- N - Name
- T - Typ
- G - Gültigkeitsbereich
- L - l-Value (Zugriff auf die Variable)
- R - r-Value (Wert der Variable)

4. Ausdrücke

Anmerkung
Beliebtes Klausuren-Thema!

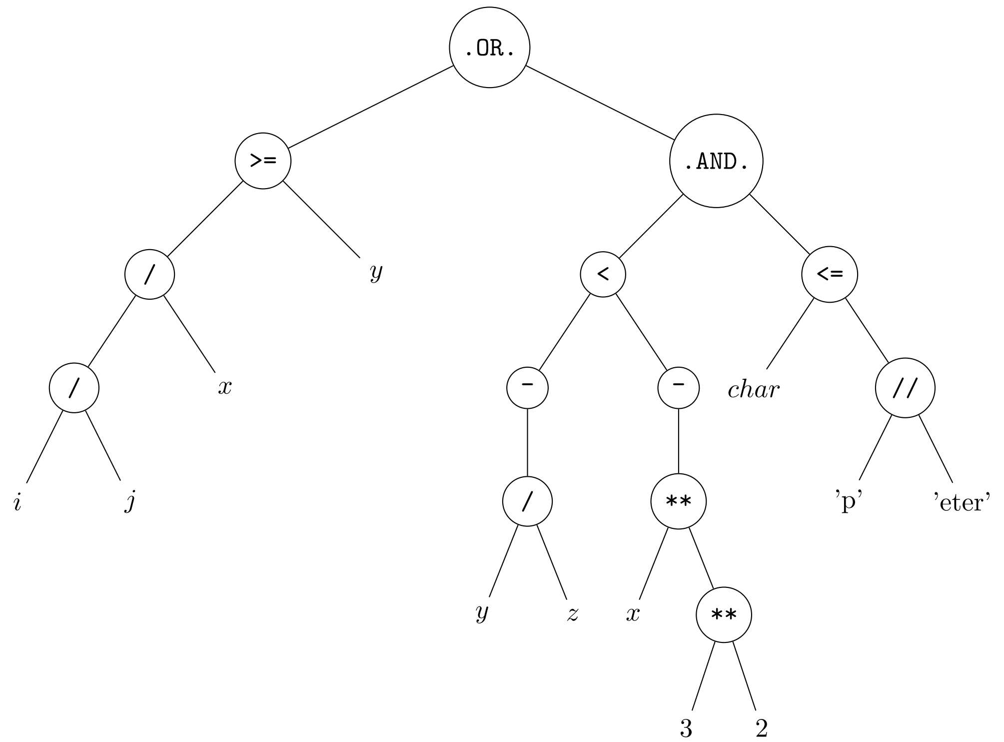
Ein Ausdruck (= expression) wird in Fortran folgendermaßen ausgewertet:

1. Konstanten und Objekte werden ausgewertet
2. geklammerte Ausdrücke werden von innen nach außen ausgewertet
3. Funktionen werden aufgerufen
4. Operatoren höherer Priorität werden vor Operatoren niedrigerer Priorität behandelt
5. sind die Prioritäten gleich, so wird von links nach rechts gearbeitet (Ausnahme: `**`)

Ein Ausdrucksbaum zeigt auf, wie in Fortran Ausdrücke ausgewertet werden: Für den Ausdruck (auch Infix-Notation)

```
logo = i / j / x >= y .OR. - y / z < - x ** 3 ** 2 .AND. char <= 'p' // 'eter'
```

sieht der Baum folgendermaßen aus:



Es gibt verschiedene Notationen um Ausdrücke aufzuschreiben:

- Präfix-Notation : Task → rekursiv linke Seite → rekursiv rechte Seite
- Infix-Notation : rekursiv linke Seite → Task → rekursiv rechte Seite

- Postfix-Notation : rekursiv linke Seite → rekursiv rechte Seite → Task

Für unseren Ausdruck bedeutet das:

- Präfix-Notation: `= logo .OR. >= / / i j x y .AND. < NEG / y z NEG ** x ** 3 2 <= char // ‚p‘ ‚eter‘`
- Postfix-Notation: `logo i j / x / y >= y z / NEG x 3 2 ** ** NEG < char ‚p‘ ‚eter‘ // <= .AND. .OR. =`

5. Unterprogramme

In Fortran gibt es 2 Arten von Unterprogrammen: Funktionen und Subroutinen. Funktionen berechnen aus übergebenen Argumenten einen neuen Wert, während Subroutinen auch Anweisungen wie `write(*,*)` ausführen können. Eine Funktion sieht in Fortran folgendermaßen aus:

```
function funcName(x,y,z)
 Anweisung1
 Anweisung2
 Anweisung3
end function funcName
```

Eine Subroutine so:

```
subroutine subName(x,y,z)
 Anweisung1
 Anweisung2
 Anweisung3
end subroutine subName
```

Der Aufruf von Funktionen oder Subroutinen geht so:

```
ergebnis = funcName(bla, bla, blub)
call subName(bla, bla, blub)
```

Die Variablen `x`, `y` und `z` sind sogenannte formale Argumente , die beim Aufruf des Unterprogramms im Hauptprogramm mit den aktuellen Argumenten `bla` und `blub` assoziiert werden. In vielen anderen Programmiersprachen (**Also nicht in Fortran!**) wird diese Assoziation als call-by-value realisiert, das heißt

- Das aktuelle Argument wird ausgewertet und das Ergebnis wird in den korrekten Typ konvertiert.
- Der Ergebniswert wird als Initialwert an das formale Argument im Unterprogramm übergeben.
- Änderungen des formalen Arguments im Unterprogramm haben **keine** Auswirkungen auf das aktuelle Argument.

Fortran assoziiert hingegen mit call-by-reference , das heißt:

- Das formale Argument wird für die gesamte Ausführungsdauer des Unterprogramms mit der Variable, die als aktuelles Argument übergeben wurde, assoziiert.
- Das heißt, dass das formale Argument ein Alias für die als aktuelles Argument übergebene Variable ist
  ⇒ für die Dauer der Ausführung des Unterprogramms haben formales Argument und aktuelles Argument denselben l-Value
  ⇒ Änderungen des formales Arguments bewirken die **gleichen** Änderungen des aktuelles Arguments.

In Fortran wird immer call-by-reference benutzt, allerdings darf das aktuelle aktuelle Argument auch ein Ausdruck sein, für dessen Ergebnis eine versteckte Variable angelegt wird, die per Referenz an das formale Argument übergeben wird. ⇒ In diesem Fall sollen keine Änderungen am formalen Argument vorgenommen werden.

Unterprogramme werden in Fortran in 4 Schritten aufgerufen:

1. Auswertung/Bestimmung des aktuellen Arguments
2. Assoziation der aktuellen Argumente mit den formalen Argumente (per Referenz)
3. Unterprogramm-Sprung
4. Rücksprung an die Aufrufstelle bei erreichen einer `return`-Anweisung oder `end` im aufgerufenen Unterprogramm

Da Fortran per call-by-reference assoziiert, gibt es verbotene Seiteneffekte (side effects). Diese dürfen `nicht` in Unterprogrammen verwendet werden.

- Veränderungen von Variablen im Ausdruck durch Funktionsauswertung in demselben Ausdruck (⇒ die Auswertung ist abhängig von der Auswertungsreihenfolge)
  `Y = X + X * F(X)` → Zuerst werden alte `X` addiert, dann wird `X` modifiziert
  `Z = F(X) * X + X` → `X` wird verändert, dann werden neue `X` addiert.
  `if(x < f(x))` → besser wäre: `y = x; if(x < f(y))`
- Assoziation mehrerer formaler Argumente mit demselben aktuellen Argument, wenn eines dieser formalen Argumente innerhalb des Unterprogramms verändert wird:

```fortran
subroutine sub(a,b)
 integer :: a, b
 b = a + b
end subroutine sub

call sub(x,x)
```

Um solche Probleme zu vermeiden kann man formale Argumente mit einen Schreib- bzw. Leseschutz ausstatten. Dafür gibt es in Fortran das sogenannte `intent`-Attribut.

```fortran
subroutine test(a, input, output)
 integer, intent(inout) :: a
 integer, intent(in) :: input
 integer, intent(out) :: output
end subroutine test
```

- `a` muss einen definierten Anfangszustand haben
- `input` hat einen Schreibschutz, es darf nur gelesen werden
- `output` hat einen Leseschutz, es darf nur geschrieben werden

Das `optional`-Attribut kann für formale Argumente benutzt werden, die nicht zwingend für die korrekte Funktionsweise des Unterprogramms notwendig sind. Im Unterprogramm muss man dann prüfen, ob dieses Argument übergeben wurde:

```
subroutine sub(a,b,opt)
 integer :: a,b
 integer, optional :: opt

 if(present(opt))
  b = a + opt
 else
  b = a
 end if
end subroutine sub
```

Ein Unterprogramm kann sich auch selbst aufrufen, wenn das Unterprogramm das Attribut `recursive` besitzt. Diese sind charakterisiert durch: Mehrere Instanzen (Aktivierungen) des rekursiven Unterprogramms sind gleichzeitig aktiv, das heißt mehrere Instanzen seines Aktivierungsblocks können gleichzeitig auf dem Laufzeitstapel liegen. Im Aktivierungsblock liegen alle Parameter, lokalen Variablen, eventuell. ich andere Informationen zur Aufrufverwaltung (z.B. Rücksprungadresse, ...); jede Aktivierungsblockinstanz ist eine vollständige Kopie mit eigenem Speicher.

```
recursive function fakultaet(n) result (res)
 integer, intent(in) :: n
 integer :: res

 if(n == 0) then
  res = 1
 else
  res = n * fakultaet(n-1)
 end if
end function fakultaet
```

```
recursive function ggt(a,b) result (g)
 integer :: a, b, g

 if(b == 0) then
  g = a
 else
  g = ggt(b, mod(a,b))
 end if
end function ggt
```

Man kann viele Funktionen und Subroutinen auch in sogenannten Modulen zusammenfassen, die dann im Hauptprogramm mit `use modulname` eingebunden werden können. Beim Kompilieren ist darauf zu achten, dass immer von "unten nach oben" kompiliert wird, also vom untersten Modul bis zum Hauptprogramm.

Ein Modul definiert in der Regel einen ADT (abstrakter Datentyp), das heißt mindestens einen öffentlichen Datentyp samt aller notwendigen Grundoperationen auf/mit Objekten dieses Typs. Häufig wird die innere Struktur der Objekte (bzw. des Typs) vor Zugriffen von außen geschützt (Datenkapselung, information/data hiding, ...), um Fehler im Umgang mit diesen Objekten zu vermeiden (z.B. inkonsistente innere Zustände, ...).

6. Benutzerdefinierte Typen

In Modulen werden häufig Datentypen vom Benutzer definiert, zum Beispiel der Datentyp `kreis`

```
type kreis
 private !kein Zugriff auf Komponenten im Hauptprogramm
 real :: radius
 real :: mitteX
 real :: mitteY
end type kreis
```

Für diesen neuen Datentyp funktionieren die alten Operatoren wie `+` nicht mehr (was soll den die Summe aus 2 Kreisen sein?). Deswegen muss man sich eine neue Addition von Kreisen ausdenken:

```
function add(kreis1, kreis2)
 type(kreis), intent(in) :: kreis1, kreis2
 type(kreis) :: add

 add%raduis = kreis1%radius + kreis2%radius
 add%mitteX = (kreis1%mitteX + kreis2%mitteX) / 2
 add%mitteY = (kreis1%mitteY + kreis2%mitteY) / 2
end function add
```

Wie man sieht kann man mit % auf die einzelnen Komponenten zugreifen. Um jetzt wirklich im Programm `kreis1 + kreis2` zu benutzen, muss man noch den Operator `+` überladen. Dazu werden generische Schnittstellen benutzt. Man kann auch das Gleichheitszeichen, also die Zuweisung `=` überladen.

```
interface operator(+)
 module procedure add
end interface operator(+)

interface assignment(=)
 module procedure gleich
end interface assignment(=)
```

Diese Operatorüberladungen brauchen immer das `intent(in)`-Attribut in den Funktionen! Überladungen von `=` brauchen dagegen sowohl `inten(in)` als auch `intent(out)`.

Mit dem `interface`-Block kann man auch lange und sperrige Namen von Funktionen einkürzen, so dass man statt `langUndSperrig(a,b)` auch `kurz(a,b)` verwenden kann:

```
interface kurz
 module procedure langUndSperrig
end interface kurz
```

7. Felder (Arrays)

Bisher hatten wir nur Skalare als Variablen. Was aber, wenn wir nicht wissen, wie viele Variablen wir brauchen werden. Dann helfen uns Felder. Felder haben eine homogene Datenstruktur, das heißt alle Elemente haben den selben Datentyp, den Elementtyp . Es gibt sowohl ein- als auch mehrdimensionale Felder (die höchste Dimension ist 15), also Vektoren, Matrizen, Tensoren, ... Der lesende als auch schreibende Zugriff erfolgt mittels ganzzahliger Indizees, also `vector(3)`, `matrix(2,5)`, `tensor(3,4,6)`, ... Unzulässige Indexwerte ermöglichen beliebige Speicherzugriffe auch außerhalb des Feldes, wenn sie nicht beim Kompilieren erkannt werden. Es kann also folgender Laufzeitfehler auftreten: `Index out of bounds`.

Kapitel IV

Ein- und Ausgabe

Anhang

Index