Bereiche der Informatik

Programmiersprache ist lexikalisch, syntaktisch und semantisch und eindeutig definiert.
Compiler = Übersetzer Programmiersprache -> Maschinensprache
Interpreter = arbeitet das Programm ab
Laufzeitsystem = stellt grundlegende Operationen und Funktionen zur Verfügung

Technische Informatik
- beschäftigt sich mit der Hardware (Konstruktion)
- Wichtige Firmen: Intel, Globalfoundries, Infineon
- Datenleitungen (Internet)

Praktische Informatik
- beschäftigt sich mit der Software (OS, Compiler, …)
- Alltägliche Software: rund 1 Fehler pro 100 Zeilen
- Wichtige Software (Raketen, OS, …) rund 1 Fehler pro 10.000 Zeilen
- Programm -> Compiler -> Maschinencode (Objectcode) -> Linker -> ausführbares Programm
  ↑
  Bibliothek

Theoretische Informatik
- beschäftigt sich mit Logik, formale Sprachen, Automatentheorie, …
- Komplexität, …

Angewandte Informatik
- beschäftigt sich mit der Praxis, Nutzer, Interaktion Mensch - Maschine, …

Maßeinheiten und Größenordnungen

Informationen und Speicher

---

bit

Kunstwort aus „binary“ und „digit“
kann nur 2 Werte speichern: 0 und 1

---

nibble

Hexadezimalziffer
bündelt 4 bits
kann 16 Werte annehmen: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Byte

bündelt 2 nibble, also 8 bit
gebräuchlichste, direkt addressierbare, kleinste Speichereinheit

ROM

„read-only-memory“
speichert wichtige Informationen, auch ohne Strom, wie Uhrzeit, Infos über Festplatte, …
nicht mehr änderbar, außer durch Belichtung, …

RAM

„random-access-memory“
Zugriff auf alle möglichen Adressen (insbesondere Hauptspeicher)

mehr Speicher

Kilo-, Mega-, Giga-, Terra-, Peta-, Exabyte
1kB = 1.000 Byte, 1MB = 1.000.000 Byte, 1 GB = 1.000.000.000 Byte
1KiB = 2^10 Byte, MiB = 2^20 Byte, 1GiB = 2^30 Byte
1kb = 1000 Bit

Zahldarstellungen für ganze und gebrochene Zahlen

Basis-Konversion ganzer Zahlen

9_10 = [1001]_2
10_10 = [1010]_2

Immer durch die Zielbasis b dividieren und den Rest als nächste Ziffer von hinten nach vorne notieren.

[57]_10 → 57/2 → 28 R 1 ⇒ letzte Ziffer der Binärdarstellung

28/2 → 14 R 0 ⇒ vorletzte Ziffer der Binärdarstellung

14/2 → 7 R 0
7/2 → 3 R 1
3/2 → 1 R 1
1/2 → 0 R 1

⇒ [111001]_2

Umgekehrte Richtung

111001 : 1010 = 101 R 111 → 5 R 7 → [57]_10
1010
——-
01000
———
010001
001010
———-
000111

von Basis 2 nach Basis 4, 8 und 16

[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

Basis-Konversion gebrochener Zahlen

Festkommadarstellung (nur Betrag der Zahl ohne Vorzeichen):
Gewichte: B^-k B^-(k+1) … B^3 B^2 B^1 B^0 . B^-1 B^-2 B^-3 … B^-l
Ziffern: m_k m_k+1 … m_-3 m_-2 m_-1 m_0 . m_1 m_2 m_3 … m_l = Summe von i=k bis l von m_i*B^-i

Konvertierung des ganzehligen Anteils vor dem „.“ wie gehabt.

Beispiel: [0.1]_3 → 1/3

Konvertierung

Wiederholt mit Zielbasis b multiplizieren und den jeweiligen ganzzahlligen Anteil als Nachkommaziffern (von links nach rechts) notieren und wegnehmen:
→ **b=2**
0.625 * 2 = **1**.25 [0.101]_2 = 5/8
0.25 * 2 = **0**.5
0.5 * 2 = **1**

→ **b=10** [0.625]_10
0.101 * 1010
———————
1010
1010
———————
110.010
0.010 * 1010 = **10**.100
0.100 * 1010 = **101**.0

→ b=2
0.1 * 2 = **0**.2 [0.0<u>0011</u>]
0.2 * 2 = **0**.4
0.4 * 2 = **0**.8
0.8 * 2 = **1**.6
0.6 * 2 = **1**.2

Probe 0.1 * 10 muss 1 sein:
0.0<u>0011</u> * 1010
———————
<u>0011</u>
<u>0011</u>
———————
0.<u>1</u> → 1.0

0.2 → [0.<u>0011</u>]
0.3 → [0.01<u>0011</u>]
0.4 → [0.011<u>0011</u>]
0.5 → [0.1]

0.6 → [0.1$\underline{0011}$]
0.7 → [0.1011$\underline{0011}$]
0.8 → [0.11$\underline{0011}$]
0.9 → [0.1110$\underline{011}$]

⇒ Problem: Rundungen schon bei 1/10 → falsche Nachkommastellen

⇒ Lösung: Gleitkommazahlen

Gleitkommazahlen, Rundungsfehler und Genauigkeitsprobleme

Gleitkommazahlen

Gleitkommazahlen = Fließkommazahlen = Gleitpunktzahlen = Fließpunktzahlen = floating-point number

Gleitkommaformat: R = R(b,l,$\underline{e}$,overline(e))
mit Basis b
mit Mantissenlänge l
mit Exponentenbereich zwischen $\underline{e}$ und overline(e)

Eine GKZ ist entweder =0 oder x = (-1)^s * m * b^e
mit Vorzeichen(bit) s Element {0;1}
mit Mantisse m = [0.m_1 m_2 m_3 … m_l]_b mit Mantissenziffern m_i Element {0;1;2;..;b-1}
mit e Element {$\underline{e}$;$\underline{e}$+1;$\underline{e}$+2;…;overline(e)}

Beispiel: R(2,3,-1,+2) → 1 bit für s, 2 bits für e, 3 bits für m
bei negativen Exponenten → Verschiebung, bis Exponent ~ 00

m=0.	111	110	101	100	11	10	1	0
e=-1	7/16	6/16	5/16	4/16	3/16	2/16	1/16	0
e=0	14/16	12/16	10/16	8/16	6/16	4/16	2/16	0
e=1	28/16	24/16	20/16	16/16	12/16	8/16	4/16	0
e=2	56/16	48/16	40/16	32/16	24/16	16/16	8/16	0

Es gibt mehrere Darstellungen für eine Zahl!
unwichtige Zahlen, die auch anders dargestellt werden können

Zahlenstrahl mit darstellbaren Werten (besonders dicht um 0, große Abstände zwischen Zahlen ab 2)

Größte darstellbare Zahl x_max = 0.111111…1 = (1-b^l)*b^overline(e)
Kleinster darstellbarer normalisierter Betrag: x_min,N = 0.10000…0 = b^$\underline{e}$-1
Kleinster darstellbarer denormalisierter Betrag: x_min,D = 0.00000…1 = b^$\underline{e}$-l

Probleme:
- absolute/relative Fehler bei Zahlen, die zwischen 2 darstellbaren Zahlen liegen
- → Rundungen bei annähernd jeder Rechnung!
- Grundrechenarten können nicht darstellbare Zahlen erzeugen

normalisierte Mantisse/GKZ: erste Mantissenziffer m_1 != 0
denormalisierte Mantisse/GKZ: mit Exponent e=$\underline{e}$ und erste Mantissenziffer m_1=0

da das erste Mantissenbit häufig eine Eins 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.

Rundung

O: reelle Zahlen (IR) → GK-Raster (R)

Rundungsregeln:
1. O(x) = x wenn x Element R
2. x,y Element IR: $x < y \Rightarrow O(x) \leq O(y)$
3. O(-x) = - O(x)

Nur manche Rundungen haben diese Eigenschaft

Rundungsmodi
- „to nearest“: zur nächstengelegenen GKZ, wenn genau in der Mitte → abwechselnd auf- und abrunden
- „trancation“: Abschneiden → betragskleiner runden
- „augmentation“: Zusatz → betragsgrößer runden
- „upward“: nach oben → in Richtung $+\infty$
- „downward“: nach unten → in Richtung $-\infty$

Gleitkomma-Arithmetik

O Element {Rundungsmodi}, o Element {+,-,*,/}
x,y Element R: $x \circledcirc y := O(x \circ y)$ $\circledcirc$…GK-Operation in R

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

Grundstrukturen von Algorithmen

Sequenz = einzelne Anweisungen hintereinander
Selektion = Verzweigung
Repetition = Wiederholung

Variablen und Daten

Am Beispiel eines Biertrinkers, der nach dem Genuss noch ein paar Besorgungen machen muss.
Variable **Durst** (LOGICAL)
Variable **Geld** (INTEGER)
Variable **PreisDerBesorgung** (INTEGER)
Variable **Rest** (INTEGER)
Variable **Bierpreis** (INTEGER)
Variable **WirtschaftAnnehmbar** (LOGICAL)
Variable **Autofahrer** (LOGICAL)
Variable **AlkoholGrenzwert** (REAL)
Variable **AlkoholVergiftungsWert** (REAL)

Schleifen

Zählschleifen

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

- Zustand der Zählvariable (=Schleifenindex) `i` vor der Schleife geht verloren (auch wenn die Schleife 0-mal läuft)
- Zählvariable `i` darf im Inneren der Schleife nicht verändert werden
- Endzustand der Zählvariable `i` nach der Schleife ist nicht definiert
- Ausdrücke (=Formel) werden zu Beginn genau genau 1-mal (vor der ersten Iteration) berechnet und sind dann fest
- Anzahl der Iterationen N = max{0,`nint[(e-a+i)/(i)]`}

Einfache Syntax

Lexikalische Eigenschaften von Fortran:

Es gibt einen zulässigen Fortran-Zeichensatz, z.B. Buchstaben A-Z, 0-9, Sonderzeichen, Operatoren,…

Lexikalische Einheiten (Symbole/Tokens):

1. Keywords: nicht reserviert → Variablen können auch so benannt werden
2. Identifiers (Namen): Länge max. 63 Zeichen → Variablennamen können Buchstaben, Zahlen und den Unterstrich beinhalten
3. Literale (Konstanten): 3 → Integer, 2.876 → Real, .TRUE. → Logical, „Hallo“ → String
4. Labels (Marken): 00000 … 99999 → Sprungmarken
5. Separatoren (Trennsymbole): ( ), /, /(/), [ ], =, =>, :, ::, ,, ;, %
6. Operatoren: +, -, *, /, **, //, ==, <=, <, /=, >, >=, .NOT., .OR., .AND., .EQV., .NEQV.

Alte Quellformen bis vor F90 (insbesondere F66, F77):

- Namen max. 6 Zeichen lang!
- Lochkarten 80 Zeichen breit

```
C Com|m| ents
* Kom|m| entare
  123 |  | PRO  GRAMM  MY PROG
99999 |  | PRODUCT
      |1 | =
      |2 |        X*Y+Z
```

```
12345|6|                                                    72|73   ...   80
```

Neue Quellformen (seit F90)

- Zeilen nun max. 132 Zeilen lang
- Kommentare beginnen mit `!` (bis Zeilenende)
- neue Zeile ⇒ neue Anweisung (außer letztes signifikantes Zeichen ist `&`)
- `&` am Zeilenende bedeutet, dass die nächste nicht-Kommentar und nicht-Leerzeile die Anweisung fortsetzt.
- Die Fortsetzung darf mit `&` beginnen
- max. 39 Fortsetzugszeilen möglich
- Leerzeichen sind signifikant (⇒ alle lexikalischen Tokens sind am Stück zu schreiben!)

- Groß- und Kleinschreibung ist nicht signifikant in Namen und Keywords

Datentypen

- Wertemenge (Speicherbedarf und -interpretation (implementierungsspezifisch))
- Konstantennotation (Literale)
- Typvereinbarung
- Operationen und Funktionen

	INTEGER[(KIND=…)]	**REAL(KIND=…)**	**COMPLEX(KIND=…)**	**LOGICAL(KIND=…)**	**CHARACTER(LEN=l, KIND=…)**
Wert	{<Z>} mindestens 1 Ziffer, höchstens unendlich	{<Z>}.{<Z>} E±{<Z>} mindestens 1 Ziffer, höchstens unendlich	(<re>,<im>) mit <re>, <im>eventuell vorzeichenbehaftete reale Konstanten	`.TRUE.` oder `.FALSE.`	einzelnes Zeichen: ‚a‘, ‚0‘ Zeichenketten: {ZK}
Wertemenge	normalerweise in 2er-Komplement: $[-2^{(l-1)},2^{(l-1)}-1]$			{`.TRUE.`, `.FALSE.`}	alle möglichen Zeichenfolgen mit l Zeichen
Operationen	`+, -, *, /` (schneidet Nachkommastellen ab), `**` (schneidet Nachkommastellen ab)	`+, -, *, /, **`	`+, -, *, /, **`	`.AND., .OR., .NOT., .EQV., .NEQV.`	

	INTEGER[(KIND=…)]	REAL(KIND=…)	COMPLEX(KIND=…)	LOGICAL(KIND=…)	CHARACTER(LEN=l, KIND=…)
wichtige Funktionen	- `SIGN(x,y)` = |x| (y≥0) oder -|x| (y<0) - `INT(x)` = schneidet Nachkommastellen ab - `FLOOR(x)` = ⌊x⌋ - `CEILING(x)` = ⌈x⌉ - `SELECTED_INT_KIND(k)` → liefert KIND-Parameter, des kleinsten INTEGER-Typs, in dem alle Zahlen mit k Stellen darstellen kann.	- `AINT(x)` = signum(x) * max{n ≤ |x|} - `REAL(x)` = konvertiert zu REAL - `SELECTED_REAL_KIND(p,r)` → liefert KIND-Parameter, mit p Ziffern in der Mantisse und r im Exponenten	- `ABS(c)` → $\sqrt{x^2+y^2}$ - `REAL(c)` → Re-Teil - `AIMAG(c)` → Im-Teil - `CONJG(c)` → x-iy		- `ICHAR(c)` → interner ganzzahliger Zeichencode - `CHAR(i)` → Zeichencode zu Zeichen - `<ZK>(a:b)` → gibt String von a-tes bis b-tes Zeichen aus - `LEN(ZK)` → Länge des Strings - `TRIM(ZK)` → liefert Zeichenkette ohne anhängende Leerzeichen - `ADJUSTL(ZK)` → Inhalt wird im String nach vorne geschoben - `REPEAT(ZK, ncopies)` → String mit ncopies-facher ZK - `INDEX, SCAN, VERIFY` → durchsucht String nach vorgegebener ZK

Integer-Division

Division	Rest der Division	Beispiele (Division)	Beispiele (Rest)
a/b := `INT(a/b)`	`MOD(a,b)` := a - (a/b) * b	8/5 → 1 (-8)/5 → -1 (-8)/(-5) → 1 8/(-5) → -1	`MOD(8,5)` → 3 `MOD(-8,5)` → -3 `MOD(-8,-5)` → -3 `MOD(8,-5)` → 3
⌊a/b⌋ = `FLOOR(a/b)` = `FLOOR(REAL(a)/REAL(b))`	`MODULO(a,b)` := a - ⌊a/b⌋ * b	(8.0)/(5.0) → 1 (-8.0)/(5.0) → -2 (-8.0)/(-5.0) → 1 (8.0)/(-5.0) → -2	`MODULO(8,5)` → 3 `MODULO(-8,5)` → 2 `MODULO(-8,-5)` → -3 `MODULO(8,-5)` → -2

Potenzieren

`2**3` → $2^3 = 8$

`2**(-3)` → `INT(2^-3)` = `INT(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` = $(2^3)^2 = 2^6 = 64$

weitere Funktionen

- `SIN(x), ASIN(x), SINH(x)`
- `COS(x), ACOS(x), COSH(x)`
- `TAN(x), ATAN(x), ATAN2(x,y)=ATAN(x/y)`
- `[COT(x)]`
- `SQRT(x)`
- `EXP(x)=` ln(x)
- `LOG10(x)`

Operator-Prioritäten

12. selbstdefinierte Operatoren unär
11. ** (Rechtsassotiativität)
10. *
9. + unär
8. + binär
7. //
6. Vergleichsoperatoren
5. .NOT:
4. .AND
3. .OR.
2. .EQV., .NEQV.
1. selbstdefinierte Operatoren binär

Variablen in imperativen Programmiersprachen

Eine Variable wird durch ein 5-Tupel (N,T,G,L,R) beschrieben:
N - Name
T - Typ
G - Gültigkeitsbereich
L - l-value (Zugriff auf Variable)
R - r-value (Wert der Variable)

Wertzuweisung:	l	=	r
	Variable		Wert

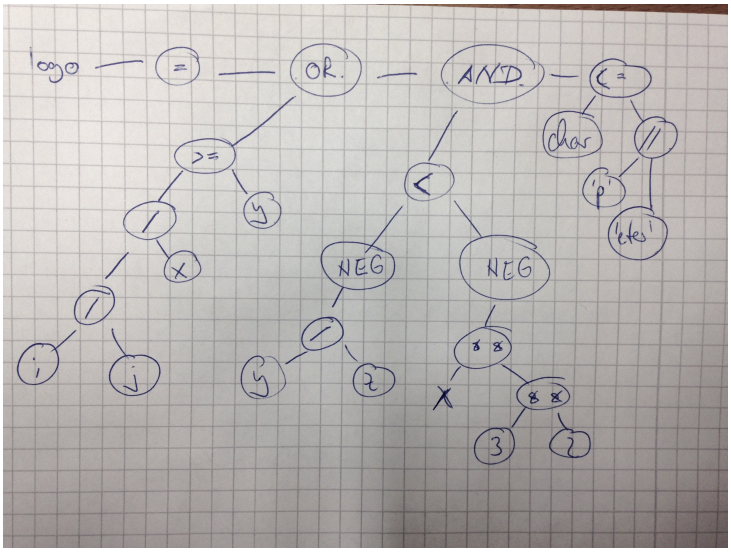
Ausdrücke (= expression)

1. Konstanten- und Objektauswertungen
2. geklammerte Ausdrücke von innen nach außen
3. Funktionsaufrufe

4. Operator höherer Priorität vor Operatoren niedrigerer Priorität
5. Operatoren gleicher Priorität von links nach rechts (Linksassoziativität, außer `**`)
6. Monadische Operatoren nicht direkt hinter dynamischen Operatoren, aber `.AND.` `.NOT.` oder statt `.AND.` geht auch `.OR.`, `.EQV.`, `.NEQV.`

Ausdrucksbaum: (allg. Syntaxbaum)
Präfix-Notation: Task → rekuriv linke Seite → rekursiv rechte Seite
Infix-Notation: rekursiv linke Seite → Task → rekursiv rechte Seite
Postfix-Notation: rekursiv linke Seite → rekrusiv rechte Seite → Task

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



Für `i=5, j=3, x=0.5, y=2.75, z=2.0, char=,p'` ⇒ `logo=.TRUE.`

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

Operandenstack für Postfix-Code

```
j = 3          x = 0.5
i = 5          1
logo           logo
```

Wiederherstellung eines Ausdrucksbaums aus einer gegebenen Notation
Vorraussetzungen:
- Unterscheidung Operator ↔ Operand immer syntaktisch/lexikalisch immer möglich
- Arität (unär ↔ binär) der Operanden muss syntaktisch bestimmt sein

aus Präfix: Leserichtung: L → R (vorwärts), Aufbau des Baums: unten → oben und L → R
aus Postfix: Leserichtung: R → L (rückwärts), Aufbau des Baums: unten → oben und R → L
aus Infix: eindeutige Interpretation nur mit Kenntnis der Operatorprioritäten möglich

Unterprogramme in Fortran
`function fname(x,y,z)` oder `subroutine subname(a,b,c)`
…
`end function` oder `end subroutine`

Aufruf mit `fname(bla, bla, blub)` oder `call subname(bla, bla, blub)`

Formales Argument = fA = Formalparameter
- in formaler Parameterliste im UP-Kopf
- wird bei UP-Aufruf assoziiert mit dem entsprechenden aktuellen Argument
- in Argumentliste des UP-Aufrufs

Parameterlänge bzw. -assoziation
In vielen Programmiersprachen als call-by-value/Wertparameter realisiert:
- zunächst Auswertung des aA-Ausdrucks ⇒ Ergebnis des korrekten Typs
- Ergebniswert wird als Initialwerten das formale Argument im UP übergeben/zugewiesen
- Änderungen des Zustandes (Werts) des fA im UP wirken sich nicht auf aA aus. (FORTRAN MACHT DAS NICHT SO!)

Oder call-by-reference/Referenzparameter:
- das fA wird für die gesamte Ausführungsdauer des UPs mit der Variable, die als aA übergeben wurde, assoziiert
- d.h. das fA ist ein Alias für die als aA übergebene Variable

⇒ für die Dauer der Ausführung des UPs haben aA und fA denselben l-Value (Speicher)

⇒ Änderungen des fAs bewirken die gleichen Änderungen des aAs

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

Aufrufmechanismus (in 4 Phasen)
1. Auswertung/Bestimmung des aA
2. Assoziation der aAe mit den fAen (per Referenz/Adresse)
3. UP-Sprung
4. Rücksprung an die Aufrufstelle bei Erreichen einer `return`-Anweisung oder `end` im Aufgerufenen UP

Verbotene Seiteneffekte (side effects):
1. Veränderungen von Größen (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 `X` modifiziert

   `Z = F(X) * X + X` → `X` wird verändert, dann werden neue `X` addiert

   `if(x < f(x))` → besser: `y = x; if(x < f(y))`
2. Assoziation mehrerer fA mit demselben aA, wenn eines dieser fA innerhalb des UPs verändert wird.

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

`intent`-Attribut
Achtung: Nur für formale Parameter erlaubt, insbesondere nicht für das Funktionsergebnis!
`intent(in)`: verlangt definierten Anfangszustand, Schreibschutz
`intent(out)`: verlangt definierten Endzustand, Leseschutz
`intent(inout)`: verlangt definierten Anfangszustand, alles erlaubt

`optional`-Attribut
Ein `optional`-fA kann beim Aufruf auch weggelassen werden (evtl. sog. keyword parameter Notation verwenden: `f(y=b, x=a)`).

Abfragefunktion (ob aA beim Aufruf übergeben wurde)
```
if(present(z)) then
   …
else
   … !kein z verwenden, da z offensichtlich nicht übergeben wurde
end if
```

Rekursion
Ein rekursives UP ruft sich selbst direkt oder indirekt auf und muss in FORTRAN als `rekursive` vereinbart werden.
Charakteristikum: mehrere Instanzen (Aktivierungen) des rekursiven UPs sind gleichzeitig aktiv, d.h. mehrere Instanzen seines Aktivierungsblocks können gleichzeitig auf dem Laufzeitstapel liegen. Im Aktivierungsblock liegen alle Parameter, lokalen Variablen, evtl. ich andere Informationen zur Aufrufverwaltung (z.B. Rücksprungadresse, …); jede Aktivierungsblockinstanz ist eine vollständige Kopie mit eigenem Speicher.

```
rekursive function rekfaculty(n) result (res)
   integer, intent(in) :: n
   integer :: res

   if(n == 0) then
      res = 1
   elese
      res = n * rekfaculty(n-1)
   if end
end function rekfaculty
```

Modulhierachie
Kompiliert wird immer von unten nach oben, also von untersten Modul bis zum HP

Ein Modul definiert i.d.R. einen ADT (abstrakter Datentyp), d.h. (mind.) einen öffentlichen Datentyp samt aller notwendigenGrundoperationen 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, …)

GGT nach dem Euklid-Algorithmus
ggt(a,b) = ggt(a-b,b) = ggt(a-2b,b) = …
ggt(a,a mod b) = ggt(a mod b,b)

```
rekursive 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
```

Benutzerdefinierte Typen
```
type my_type
   [private] !Zugriff innerhalb des Moduls immer, außerhalb nicht bei private
   <Komponentendeklaration>
end type my_type
```

Generische Schnittstellen

```
1. interface <generic_name>
      module procedure <proc_name>, … !egal ob function oder subroutine
   end interface
2. interface operator (+,-,*,/,<,…)
      module procedure <function_name>, … !intent(in)-Parameter
   end interface
3. interface assignment (=)
      module procedure <subr_name>, … !intent(out) und intent(in)
   end interface
```

Spezifische Schnittstelle definiert durch:

```
interface
   function f(x)
      integer :: a,b
      …
   end function f
end interface
```

Felder (Arrays)

- bisher nur Skalare (=Nicht-Felder bzw. 0-dimensionale Felder)
- homogene Datenstruktur, d.h. alle Elemente haben denselben Datentyp = Elementtyp
- ein- oder mehrdimensional (Vektor, Matrix, Tensor, …)
- geometrisch wie Strecke, Rechteck, Quader, …
- lesender + schreibender Zugriff auf Feldelemente mittels ganzzahliger Indizes: vec(3), vec(i+j-1), mat(2,5), **beginnt bei 1**
- effizienter, direkter Zugriff auf Elemente im Feld
- Unzulässige Indexwerte ermöglichen, wenn sie nicht zur Laufzeit (oder auch bei Kompilieren) erkannt werden, beliebige Speicherzugriffe auch außerhalb des Felds
  → Compiler-Option, um Indextests zu generieren
  ⇒ es kann der Laufzeitfehler auftreten: `Index out of bounds`

Feldtyp charakterisiert durch:

- Elementtyp (beliebig wählbarer skalarer Typ)
- Rang (= rank): Anzahl der Dimensionen, maximal 15 Dimensionen

Geometrische Gestalt (= shape) eines Feldes kann entweder statisch oder dynamisch definiert werden, und zwar durch Ausdehnungen (= extents) in jeder einzelnen Dimension.

Anzahl der Elemente in i-ter Dimension: `SIZE(Array, i) = u-l+1`
`LBOUND(Array, i)` kleinster Indexwert in i-ter Dimension
`UBOUND(Array, i)` größerer Indexwert in i-ter Dimension
`SIZE(A)` = $\prod_{i=1}^{r} SIZE(A,i)$ = Gesamtzahl aller Elemente
`SHAPE(A)` → `(/SIZE(A,i), SIZE(A,2), … , SIZE(A,r)/)`

Feldkonstruktor: `(/e_1, e_2, … , e_k/)` oder (seit FORTRAN 2003) `[e_1, e_2, … , e_k]`
→ 1-dimensionales Feld (immer), aber es gibt eine `RESHAPE`-Funktion
Bsp: `(/1,3,5,7,9/)` = `(/(i, i=1,9,2)/)` = `(/(2*i+1, i=0,4)/)`

Speicherreihenfolge: in FORTRAN spaltenweise (colum-major) → Matrix als „Bild“, d.h. 1. (vorderer) Index läuft immer am schnellsten, letzter Index am langsamsten

```
real, dimension(3,2) :: A !statisch
real, dimension(:,…,:), allocatable :: B !dynamisch, r-mal „:“

read(*,*) A !zuerst alle Zahlen erster Spalte, dann zweiter Spalte, …
```

```
read(*,*) (A(i,:), i=1,3) !zeilenweises Einlesen
do i=1,3 !zeilenweises Einlesen
   read(*,*) A(i,:) !(A(i,j), j=1,2)
end do
```

Vordefinierte Matrixfunktionen

- `SUM(A,1)` → Summation einzelner Spalten, Ergebnis ist ein Vektor
- `SUM(A,2)` → Summation einzelner Zeilen, Ergebnis ist ein Vektor
- `SUM(A)` → Summe der Summe von Zeilen, einfach alles summieren
- `PRODUCT` → analog zu SUM
- `ALL(A,[i])` → logisches UND (z.B. `ALL(A==B)`: `A==B` generiert eine Matrix mit logischen Einträgen, ob ein Element aus A und ein Element aus B identisch sind, `ALL` überprüft diese)
- `ANY(A,[i])` → logisches ODER
- `TRANSPOSE(A)` → transponiert Matrix
- `DOT_PRODUCT(v,w)` → $\sum_{i=1}^{n} v_i \cdot w_i$ (Skalarprodukt eines Vektors)
- `MATMUL(A,B)` → normale Matrixmultiplikation
- `MATMUL(A,w)` → Matrixmultiplikation mit einem Vektor
- `MATMUL(v,B)` → Vektor * Matrix

Subarrays (Teilfelder)

Indexmenge definiert durch

- Indextripel: `a:e[:s]`
- Indexvektor: `v([4,2,1,5,4])` (also (v_4,v_2,v_1,v_5,v_4)) `= [1,2,3,4,5]`

Wenn derselbe Indexwert (in einemIndexvektor) mehrmals vorkommt, darf auf das Teilfeld nur lesend zugegriffen werden. Indexmenge kann beliebig viele, also auch 1 oder 0 Indexwerte enthalten. Dies ändert nichts daran, dass diese Dimension des Teilfeldes zu ihrem Rang beiträgt, d.h. 1×5-Matrix ≠ 5-Vektor ≠ 5×1-Matrix oder 0×5-Matrix ≠ 0-Vektor ≠ 5×0-Matrix <- leere Felder d.h. diese sind nicht gestaltkonform.

Beispiel: A ist Quader (5 × 5 × 5) ⇒ r_A = 3

Teilfeld: `A(3,1:5:2,(/4,1,2/))` → Aus einem Quader wird die 3. Schicht genommen und in dieser Matrix die 1., 3. und 5. Zeile mit der 4., 1. und 2. Spalte ⇒ 3×3-Matrix mit r_A = 2

Diagonale einer quadratischen Matrix als Vektor: `(/(A(i:i),i = 1,size(A,1))/)`

Speicherplatz 4	Speicherplatz 7		Speicherplatz 1	
Speicherplatz 5	Speicherplatz 8		Speicherplatz 2	
Speicherplatz 6	Speicherplatz 9		Speicherplatz 3	

Felder mit Prozedurparametern (f.A.)

formales Argument kann sein

- statisches Feld (feste Gestalt)
- Feld übernommener Gestalt (assumend-shape array), welche durch die Gestalt des aktuellen Arguments durch die Parameterassoziation (per Referenz) festgelegt ist.

Felder als Funktionsergebnis (gilt nur für Funktionsergebnisvariable)

Indexbereiche (→ Gestalt) müssen zum Aufrufzeitpunkt der Funktion berechnet werden können; Indexgrenzen können beliebige Integer-Ausdrücke sein, die von den Werten und Eigenschaften der aAe und globalen Variablen oder Konstanten abhängen.

Automatische Felder (nur lokale Variablen im UP)
Lokale Feldvariablen, die wie ein Funktionsergebnis deklariert werden müssen, d.h. die Indexgrenzen müssen zum Aufrufzeitpunkt berechenbar sein.

Felder als Typkomponenten (in selbstdefinierten Typen)
Entweder statisch oder dynamisch mit POINTER-Attribut.
Wichtige Regel: Keine `allocatable`-Felder als
- formale Argumente
- Feld-Ergebnis einer Funktion
- Typkomponenten

```
function fun(v,w)
   real,dimension(:), intent(in) :: v,w
   real,dimension(size(v),size(w)) :: fun
   integer :: i,k

   do i = 1, size(v)
      do k = 1, size(w)
         fun(i,k) = v(i) * w(k)
      end do
   end do
end function fun

function fun_effizient(v,w)
   real,dimension(:), intent(in) :: v,w
   real,dimension(size(v),size(w)) :: fun
   real,dimension(size(v),size(w)) :: A,B

   A = spread(source=v, dim=2, ncopies=size(w))
   B = spread(source=w, dim=1, ncopies=size(v))
   fun = A * B !elementweise Multiplikation
end function fun_effizient
```

wichtige Funktionen
- `forall( i=1:n, k=1:m)`
- `deallocate` - nicht vergessen

PURE Prozeduren
- können in `forall`-Schleifen verwendet werden
- alle intrinsischen Funktionen sind pure
- keine Standard-Subroutine ist pure (außer `MVBITS`)

```
integer, save :: counter = 0
```

`save`-Attribut ist bei Initialisierung von Variablen impliziert, d.h. eine Initialisierung in der Typdeklaration macht die Variable automatisch statisch (Lebensdauer = gesamte Programmlaufzeit).

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.

ELEMENTAL Prozeduren
- müssen pure sein
- nur skalare Datenobjekte ohne `Pointer` oder `allocatable`
- viele intrinsischen Funktionen sind elemental

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.

Beispiel: `sin` ist eine elementare Funktion. `sin(A)` liefert dann also die Matrix `A` mit $\begin{pmatrix} \sin a_{11} & \dots & \sin a_{1n} \\ \cdots & & \cdots \\ \sin a_{m1} & \dots & \sin a_{mn} \end{pmatrix}$. Der Sinus wird also *elementweise* angewendet.

RESHAPE-Funktion

```
integer,dimension(7) :: integervector = (/(i,i=1,13,2)/)
```

$(1 \quad 3 \quad 5 \quad 7 \quad 9 \quad 11 \quad 13)$

```
reshape(source=integervector, shape=(/2,3/))
```

$\begin{pmatrix} 1 & 5 & 9 \\ 3 & 7 & 11 \end{pmatrix}$

```
reshape(source=integervector, shape(/5,3/), pad=(/(i+13,i=1,8)/),
order=(/2,1/))
```

$\begin{pmatrix} 1 & 3 & 5 \\ 7 & 9 & 11 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \\ 19 & 20 & 21 \end{pmatrix}$ Wenn `integervector` abgearbeitet ist, wird mit `pad` weitergemacht

Diagonale einer Matrix nicht als Teilfeld ansprechbar. `(/(imat(i,i), i=1, min(ubound(imat,1), ubound(imat,2)))/)` nur lesender Zugriff.

wichtige intrinsische Funktionen
- `LBOUND(A,2)`
- `UBOUND(A)`
- `SIZE(A,1)` → $u_1 - l_1 + 1$
- `SIZE(A)` → Anzahl aller Elemente
- `SHAPE(A)` → $[u_1 - l_1 + 1, u_2 - l_2 + 1, \ \dots]$
- `SUM(A,i)` → Feld mit Rang r-1
- `SUM(A)` → Summe aller Elemente von A
- `PRODUCT(A,i)` → Feld mit Rang r-1
- `PRODUCT(A)` → Produkt aller Elemente von A
- `ANY` logisches Oder
- `ALL` logisches Und
- `DOT_PRODUCT(v,w)` → Skalarprodukt von v und w
- `MATMUL(A,B)` oder `MATMUL(A,w)` oder `MATMUL(v,B)`
- `MINVAL(array=A, dim=2, mask=A>0)`
- `MAXVAL(array=A, dim=2, mask=A>0)`

Ein- und Ausgabe
Aufgabe: Verwaltung von Dateien (files), Datentransfer zwischen Daten und internem (Haupt-)Speicher

Lesen: Datei → Hauptspeicher
Schreiben: Hauptspeicher → Datei
Datei: normalerweise externe Datei, d.h. liegt nicht im Hauptspeicher
- eine Ansammlung von Daten auf einem externen Speichermedium
- falls explizit gewünscht kann eine Datei auch intern sein, d.h. im Speicher, dafür wird eine Zeichenkettenvariable als Datei verwendet

Datensatz: Zeile einer Textdatei oder Datensatz einer Binärdatei, engl. record,
- formatiert: Daten sind Sequenzen von Zeichen
- umformatiert: Daten sind binär, wie im Hauptspeicher dargestellt

Dateizugriff:
- sequenziell: lineare Anordnung der Daten(sätze), zu jedem Zeitpunkt gibt es eine aktuelle Position in der Datei, nur sequenzieller Zugriff von BOF → EOF
- direct access: Zugriff direkt über die Datensatznummer i (engl. Record number) auf den i-ten Datensatz

Dateiverwaltung

```
OPEN + CLOSE
BACKSPACE
REWIND      } (UNIT=u, IOSTAT=iovar, ERROR=errvar)
ENDFILE
```

`BACKSPACE`: 1. Mal - positioniert vor dem aktuellen Datensatz
2. Mal - positioniert vor dem vorherigen Datensatz
→ möglichst vermeiden

`REWIND`: setzt die aktuelle Dateiposition an/vor den Anfang der Datei (BOF)

`ENDFILE`: hängt an die aktuelle Datei hinter den aktuellen Datensatz ein EOF-Datensatz an

I/O unit:
- ganze, nichtnegative Zahl zur Identifikation einer externen Datei
- `*` zur Identifikation einer ganz bestimmten Datei, typischerweise Tastatur (= 5) oder Bildschirm (= 6)
- Zeichenkettenvariable als interne Datei

Für explizite Verbindung mit einer externen Datei ist `OPEN(UNIT=133,…)` notwendig!

Printing/Drucken in Fortran:
Zeilenvorschub wird durch 1 Zeichen einer Zeile definiert:
- „ “ → nächste Zeile
- „0“ → übernächste Zeile
- „1“ → neue Seite beginnen
- „+“ → Zeile überschreiben

Prinzip: Das, was mit einem bestimmten expliziten Format geschrieben wurde, kann mit dem selben Format wieder eingelesen werden. Ausnahme: keine Verwendung von ZK-„Konst“ oder A-Editdesktribtoren ohne Feldbreite (da diese für die Eingabe nicht möglich sind).

→ Es gibt für die aktuelle Zeile (Datensatz) immer ein ZK-Puffer im Speicher.

Formatliste wird „normal“ in Leserichtung abgearbeitet (mit Wiederholungsfaktoren, Unterlisten, …). Steuerung des Datentransfers wird durch die Formatliste (nicht vorrangig die I/O-Liste) durchgeführt!

Hollerith-Konstante (alter Stil)
- definiert eine Zeichenkette: nHz_1z_2z_3…z_n

- häufig in alten Formatangaben: `FORMAT(1H` bzw. `1HØ` bzw. `1H1` bzw. `1H+)`