Einführung in die Programmiersprache Julia
Vorlesung Computerphysik
Sommersemester 2018
Ralf Bulla
Universität zu Köln

1 Einstieg

Das erste Programm:

```
a = 1
println(a)
```

Ein Programm ist eine Abfolge von Befehlen, die nacheinander ausgeführt werden, hier:

1. `a = 1`: der Variablen `a` wird der Wert 1 zugewiesen;
2. `println(a)`: der Wert der Variablen `a` wird ausgegeben;

Etwas komplizierter:

```
a = 1
b = 2
c = a + b
a = b
b = 1 + a
println("a = ",a)
println("b = ",b)
println("c = ",c)
```

(Zur besseren Übersicht wurde hier im Argument des `println`-Befehls noch der String (Zeichenkette) `a =` etc., hinzugefügt; mehr zu `println` siehe Kap. *.) Dies erzeugt die folgende Ausgabe:

```
a = 2
b = 3
c = 3
```

Wichtig ist die Unterscheidung zwischen der Zuordnung `=` im Programm, z.B. `a = b`, und dem mathematischen Gleichheitszeichen, wie in $a^2 + a = 1$. Hier ein Beispiel:

```
a = 1
a = a + 1
b = 2
b = a + b
b = 1 + a
println("a = ",a,", b = ",b)
```

In Zeile 2 wird der Variablen `a` der Wert `a+1` zugeordnet. Hier wird zuerst die rechte Seite berechnet (ergibt den Wert 2), danach erfolgt die Zuordnung, d.h. `a` erhält den Wert 2, usw. Die Zuordnung `a = a + 1` wird im Programm gerade *nicht* als mathematische Gleichung $a = a + 1$ interpretiert, was der Aufgabe entsprechen würde, die Gleichung $a = a + 1$ nach a aufzulösen (dazu gibt es in diesem Fall offensichtlich keine Lösung).

Andererseits lässt sich die Gleichung

$$a + 1 = 2$$

nach a auflösen, im Programm kann man jedoch nicht `a + 1` den Wert `2` zuordnen, der Befehl `a + 1 = 2` liefert eine Fehlermeldung.

2 for-Schleifen

Ein einfaches Beispiel: Zu berechnen ist die Summe

$$S = \sum_{n=1}^{M} n \ ,$$

deren Wert sich auch analytisch bestimmen lässt:

$$S = \sum_{n=1}^{M} n = \frac{1}{2} M(M+1) \ .$$

(im Gegensatz zu komplizierteren Beispielen wie $S = \sum_{n=1}^{M} \sqrt{n}$.)

Die erste Variante (für $M = 5$) ist nicht besonders elegant:

```
s = 1 + 2 + 3 + 4 + 5
println(s)
```

Ebenso die folgende Variante:

```
s = 0
s = s + 1
s = s + 2
s = s + 3
s = s + 4
s = s + 5
println(s)
```

Hier ist die Verwendung einer Schleife sinnvoll:

```
s = 0
for n = 1:5
    s = s + n
end
println("Summe = ",s)
```

(hier bietet sich auch die abkürzende Schreibweise `s += n` anstelle von `s = s + n` an.) In diesem Beispiel ist dies eine sogenannte for-Schleife; die Struktur ist

```
for n = a:b
    block
end
```

und ist folgendermaßen zu lesen: Von $n = a$ bis $n = b$ führe die in `block` angeführten Befehle aus. Die for-Schleife wird durch `end` abgeschlossen. Die Zählvariable n wird dabei in jedem Schritt um 1 erhöht, wie man in folgendem Beispiel sieht:

```
for n = 3:12
    println(n)
end
```

Hier werden nacheinander alle Zahlen von 3 bis 12 ausgegeben.

For-Schleifen kommen immer dann zum Einsatz, wenn von vornherein klar ist, wie oft die Schleife durchlaufen werden soll, im Gegensatz zu while-Schleifen, siehe Kap. *.

Der Ausdruck `1:5` ist ein sog. *Range*-Objekt und entspricht in diesem Beispiel einer Liste mit allen natürlichen Zahlen von 1 bis 5. Das Beispiel von oben lässt sich auch so schreiben:

```
s = 0
coll = 1:5
for n in coll
   s += n
end
```

Die Variable `coll` enthält also die entsprechende Liste. Hier wurde auch die Schreibweise `n in coll` anstelle von `n = coll` verwendet. Die beiden Befehle `in` und `=` sind äquivalent, jedoch ist `in` eher im Sinne von $\in$ zu verstehen.

Die for-Schleife in Julia erlaubt die Iteration über die Elemente verschiedener Arten von Listen, zum einen Variationen des *Range*-Objekts, z.B.:

- `0:2:10` $\to \{0, 2, \ldots, 10\}$,
- `10:-1:0` $\to \{10, 9, 8, \ldots, 0\}$,

aber auch Arrays (Felder, siehe Kap. 3) wie in folgendem Beispiel:

```
coll = [3,7,12]
for n in coll
   println("$n*$n = $(n^2)")
end
```

Dieses Programm berechnet die Quadratzahlen n^2 für $n = 3, 7, 12$ und erzeugt die Ausgabe:

```
3*3 = 9
7*7 = 49
12*12 = 144
```

(Hinweis: Hier wurde eine andere Schreibweise im Argument des println-Befehls gewählt: mit `$x` bzw. `$(x)` wird der Zahlenwert von x an entsprechender Stelle ausgegeben.)

Die Berechnung der Summe $\sum n$ zu Beginn dieses Kapitels ist als Motivation zur Verwendung von for-Schleifen gedacht – wie in den Beispielen aus der Vorlesung und den Übungen deutlich wird, können verschiedenste Iterationen mit Hilfe von for-Schleifen realisiert werden.

Andererseits kann man zur Berechnung von Summen auch die vordefinierte Funktion `sum` verwenden:

- `sum(1:M)` berechnet die Summe $\sum_{N=1}^{M} n$;
- `sum(coll)` berechnet die Summe über alle Elemente der Liste `coll`, z.B. `sum([3,7,12])` ergibt 22;
- Die Summe

$$\sum_{N=1}^{M} f(n) \ , \quad \text{mit } f(n) = \sqrt{n}$$

lässt sich folgendermaßen berechnen:

```
f(n) = sqrt(n)
M = 5
s = sum(f,1:M)
println("Summe = ", s)
```

(zur Definition der Funktion `f(n)` in der ersten Zeile, siehe Kap. 5, Funktionen.)

Die Berechnung von Produkten lässt sich analog mit Hilfe des Befehls `prod` durchführen.

3 Felder

Als Beispiel betrachten wir die Umrechnung einer Dualzahl, z.B. $[1011]_2 = [z_4 z_3 z_2 z_1]_2$, in die entsprechende Dezimalzahl. Es gilt:

$$M = \sum_{i=1}^{n} z_i 2^{i-1} \ . \tag{1}$$

Die einzelnen Stellen der Dualzahl lassen sich zwar folgendermaßen darstellen:

```
z_1 = 1; z_2 = 1; z_3 = 0; z_4 = 1;
```

(Hier sind die einzelnen Befehle durch Semikolon getrennt in einer Programmzeile geschrieben.) Diese Darstellung ist jedoch ziemlich unhandlich, da sich die Summe in Gl. (1) so nicht als for-Schleife schreiben lässt, sondern lediglich als:

```
 M = z_1 + z_2*2 + z_3*2^2 + z_4*2^3
```

Hier bietet es sich an, die Dualzahl als Feld (engl. *array*) zu speichern. Das folgende Programm definiert ein solches Feld und gibt die Einträge der Reihe nach aus:

```
z = [1,1,0,1]
for n = 1:4
    println("z[",n,"]=",z[n])
end
```

Der Befehl `z = [1,1,0,1]` definiert also ein Feld mit vier Einträgen, auf die über `z[1]`, `z[2]`, etc. zugegriffen werden kann.

Hinweis: der erste Eintrag des Felds hat in Julia den Index 1, im Gegensatz zu vielen anderen Programmiersprachen, bei denen die Indizes mit 0 beginnen.

In vielen Fällen ist es sinnvoll, das Feld zuerst zu definieren und dann zu belegen. In folgendem Beispiel wird mit `z = Array{Int64}(L)` ein Feld der Länge L definiert und in einer for-Schleife mit $z_i = i^2$ belegt:

```
L = 5
z = Array{Int64}(L)
for n = 1:L
   z[n] = n*n
   println("z[",n,"]=",z[n])
end
```

`Int64` im Argument von `Array` bedeutet, dass ein Feld mit ganzen Zahlen (engl. *integer*) als Einträgen definiert wird, in diesem Fall ganze Zahlen mit 64 bits (mehr zu Datentypen in Kap. *).

Die Umrechnung von $[1011]_2$ in die entsprechende Dezimalzahl lässt sich damit folgendermaßen durchführen:

```
z = [1,1,0,1]
s = 0
for n = 1:4
   s += z[n]*2^(n-1)
end
```

Julia enthält einige nützliche Befehle zur Definition und Belegung von Feldern. Hier eine Auswahl:

- `zeros(N)` erzeut ein Feld der Länge N, wobei alle Einträge $= 0$ gesetzt werden. Die Einträge sind vom Typ Float64, andere Datentypen lassen sich mit `zeros(type,N)` festlegen, z.B. `zeros(Int64,N)`.

- `ones(N)`: analog zu `zeros(N)`; hier werden die Einträge $= 1$ gesetzt.

- `Array(n:m)` erzeugt ein Feld der Länge $m - n + 1$ (für $m \geq n$) mit den Einträgen $n, n+1, \ldots, m$; falls `m` und `n` beide vom Typ Int64 sind, ist das Feld vom Typ `Array{Int64,1}`.

- `x = linspace(a,b,N)` erzeugt ein Range-Objekt mit N linear verteilten Werten zwischen $x_1 = a$ und $x_N = b$, d.h.

  $$x_i = a + (b-a)\frac{i-1}{N-1} \ .$$

  Der Typ ist `LinSpace{Float64}`, `x` kann aber in der weiteren Rechnung wie ein Feld verwendet werden (oder mit `Array(x)` in ein Feld umgewandelt werden).

- `y = logspace(a,b,N)` (analog zu `linspace(a,b,N)`) erzeugt ein Range-Objekt mit N logarithmisch verteilten Werten zwischen $y_1 = 10^a$ und $y_N = 10^b$, d.h.

  $$\log_{10} y_i = a + (b-a)\frac{i-1}{N-1} \ .$$

- `rand(N)` erzeugt ein Feld mit N gleichverteilten Zufallszahlen x_i im Intervall $[0,1[$ vom Typ Float64. `rand(coll,N)` erzeugt ein Feld mit N zufällig aus der Liste `coll` ausgewählten Einträgen (z.B. `rand(1:6,10)`). Der Typ entspricht dem der Elemente von `coll`. Hinweis: `rand()` erzeugt eine einzelne Zufallszahl im Intervall $[0,1[$ vom Typ Float64, `rand(1)` jedoch ein Feld der Länge 1 (analog `rand(coll)` und `rand(coll,1)`).

Zur Bearbeitung bereits definierter Felder gibt es eine Reihe nützlicher Werkzeuge – hier eine Programmbeispiel:

```
a = [3,5,7,9,10]
b = a[2:4]          # b = [5,7,9]
push!(a,20)         # a = [3,5,7,9,10,20]
c = a[5:end]        # c = [10,20]
append!(c,b)        # c = [10,20,5,7,9]
pop!(b)             # b = [5,7]
sort!(c)            # c = [5,7,9,10,20]
```

In diesem Programm wird zunächst mit `a = [3,5,7,9,10]` das Feld `a` mit fünf Einträgen definiert.

- `b = a[2:4]`: hier wird das Feld `b` mit den Einträgen 2 bis 4 von `a` definiert, `b` hat also die Einträge 5,7,9; `end` steht für den letzten Eintrag, siehe weiter unten `c = a[5:end]`; mit `a[1:2:5]` kann man aus `a` die Einträge No. 1,3 und 5 herausschneiden.
- `push!(a,20)` fügt dem Feld `a` am Ende einen weiteren Eintrag (hier 20) hinzu; es lassen sich auch mehrere Elemente hinzufügen, z.B. `push!(a,1,2,3)`. Hinweis: Funktionen mit ! am Ende des Funktionsnamens ändern das erste Objekt in der Liste der Argumente.
- `append!(c,b)`: hier werden dem Feld `c` alle Einträge des Felds `b` am Ende hinzugefügt.
- `pop!(b)`: das letzte Element von `b` wird herausgenommen; mit `z = pop!(b)` wird die Variable z mit diesem Element belegt.
- `sort!(c)`: die Elemente von c werden nach aufsteigender Reihenfolge sortiert.

4 if-Verzweigung

Als Beispiel für eine Verzweigung betrachten wir die Subtraktionsmethode zur Umwandlung einer Dezimalzahl M in die entsprechende Dualzahl. Für $M = 41$ gilt $[41]_{10} = [101001]_2$, mit der Anzahl an Stellen $n = 6$. Im ersten Schritt der Subtraktionsmethode wird die Differenz $M - 2^{n-1}$ gebildet, in diesem Beispiel also $41 - 2^5 = 9$. Das Ergebnis ist ≥ 0, damit wird die sechste Stelle der Dualzahl mit 1 belegt: $z[6] = 1$. Ist die Differenz $M - 2^{n-1} < 0$, wird die entsprechende Stelle mit Null belegt.

Damit haben wir bereits ein Beispiel für eine Verzweigung, hier das Programm:

```
M = 41
n = 6
z = Array{Int64}(n)
if (M-2^(n-1)) >= 0
   z[n] = 1
else
   z[n] = 0
end
println(z[n])
```

Diese if-Verzweigung hat die folgende Struktur:

```
if Bedingung
    block 1
else
    block 2
end
```

Falls die Bedingung wahr ist, z.B. `9 >= 0`, wird `block 1` ausgeführt im anderen Fall wird `block 2` ausgeführt.

5 Funktionen

Falls in einem Programm mehrmals die Funktionswerte einer Funktion $f(x)$ berechnet werden sollen, ist es zweckmäßig, die mathematische Funktion $f(x)$ als Funktion `f(x)` (Funktion hier im Sinne einer Programmstruktur) zu definieren. Für $f(x) = x^2 + 2x + 3$ geht das sehr einfach mit dem Befehl

```
  f(x) = x^2 + 2*x + 3
```

Die Funktionswerte können im Programm mit `f(1), f(2.5), f(sqrt(2))`, etc. aufgerufen werden. Das folgende Programm gibt die Funktionswerte $f(n)$ für $n = 1, 2, \ldots, 10$ aus:

```
f(x) = x^2 + 2*x + 3
for n = 1:10
  println("f($n) = $(f(n))")
end
```

Diese kompakte Schreibweise zur Definition der Funktion `f(x)` ist äquivalent zu:

```
function f(x)
  y = x^2 + 2*x + 3
  return y
end
```

Auch hier werden die Funktionswerte mit `f(1)` etc. aufgerufen.

Diese Definition einer Funktion hat folgende Struktur:

```
function fname(arglist)
  body
  return value
end
```

- `fname` ist der Funktionsname unter dem die Funktion im Programm mit `fname(...)` aufgerufen wird.
- `arglist` ist die Liste der Argumente der Funktion (eines oder mehrere, siehe unten).
- In `body` wird die Berechnung des Funktionswertes durchgeführt. Diese Berechnungen können natürlich aufwändiger sein als in obigem Beispiel.
- Nach dem Befehl `return` steht in `value` der auszugebende Funktionswert.

Die Verwendung von Funktionen ist generell nützlich, um ein Programm übersichtlicher zu strukturieren. So können Programmteile in Funktionen ausgelagert werden, wie in folgendem Beispiel gezeigt (Umwandlung von Dezimalzahlen in Dualzahlen mit Hilfe des Subtraktionsalgorithmus):

```
function dec2dual(M)
  n = Int(floor(log2(M) + 1))
  z = zeros(Int64,n)
  for i = n:-1:1
    L = M - 2^(i-1)
    if L >= 0
      z[i] = 1
      M = L
    end
  end
  return z
end

M_values = [2,7,11,16]

for M in M_values
  z = dec2dual(M)
  println(M,"\t",z)
end
```

In der Funktion `dec2dual(M)` wird zunächst ein Feld `z` geeigneter Länge definiert (hier mit Nullen vorbelegt), dann wird in der for-Schleife der Subtraktionsalgorithmus ausgeführt. Das Argument der Funktion ist also die Dezimalzahl `M`, ausgegeben wird mit `return z` das Feld `z` mit den Stellen der Dualzahl.

Damit die Funktion auch ausgeführt wird, muss diese im Programm aufgerufen werden, dies geschieht hier mit `z = dec2dual(M)`. Das Programm gibt also für die in `M_values` gespeicherten `M`-Werte jeweils das entsprechende Feld `z` aus.

Hier ein Beispiel für eine Funktion mit mehreren Argumenten:

```
f(x,y,z) = sqrt(x^2 + y^2 + z^2)
```

Der Funktionsaufruf erfolgt hier mit `f(1,1,2)`.