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.) 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 = 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 (while-Schleifen werden in Kap. * behandelt). 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 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.

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

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

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

Hier werden die Quadratzahlen n^2 für $n = 3, 7$ und 12 ausgegeben.

3 Felder

Als Beispiel betrachten wir die Umrechnung einer Dezimalzahl, 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;
```

Dies 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 = 1:5
   s += z[n]*2^(n-1)
end
```

Julia enthält einige nützliche Befehle zur Definition un 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.

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