Programmieren im Bauingenieurwesen

Einführung und Grundlagen

Autoren
Sebastian Pech, Raphael Suda
Datum
23. September 2021
Erstellt mit Julia
1.6.1
Version
cb4aa5b

Inhaltsverzeichnis

1 Kontrollstrukturen

Der Code in den bisher behandelten Beispiele setzte sich aus einer sich nicht-verzweigenden Verkettung von Befehlen zusammen. Für sehr rudimentäre Programme mag das ausreichen, in den meisten Fällen werden aber Verzweigungen in Form von `if`-Bedingungen und Schleifen benötigt.

1.1 Verzweigungen

Dieses Konstrukt erlaubt es, abhängig von einer *Bedingung* (die ausgewertet vom Typ `Bool` sein muss) einen Codeteil auszuführen oder zu überspringen. Die einfachste Form ist das `if`-Statement:

```
x = -1
if x < 0
    println("x ist negativ.")
end
```

```
x ist negativ.
```

In diesem Beispiel ist die *Bedingung* `x < 0`. Wie bereits im Abschnitt zu den Datentypen gezeigt, ergibt die Auswertung dieses Ausdrucks einen Wert vom Typ `Bool`:

```
julia> x < 0
true

julia> typeof(x < 0)
Bool
```

Die zweite Form ist die Kombination aus `if` und `else`. `else` bezeichnet den Zweig, der ausgeführt wird, falls die Bedingung nicht erfüllt ist.

```
x = 1
if x < 0
    println("x ist negativ.")
else
    println("x ist positiv.")
end
```

```
x ist positiv.
```

Da `x < 0` ausgewertet `false` ergibt, wird der `if`-Zweig übersprungen und der `else`-Zweig ausgeführt.

Die dritte Form ist im Grunde eine vereinfachte Schreibweise für Kombinationen von Form eins und Form zwei. Der folgende Code

```
x = 9
if x < 0
    println("x ist negativ.")
else
    if x <= 10
        println("x ist eine Zahl von 0 bis 10.")
    else
        println("x ist größer als 10.")
    end
end
```

```
x ist eine Zahl von 0 bis 10.
```

kann einfacher und übersichtlicher durch `elseif` geschrieben werden.

```
x = 9
if x < 0
    println("x ist negativ.")
elseif x <= 10
    println("x ist eine Zahl von 0 bis 10.")
else
    println("x ist größer als 10.")
end
```

```
x ist eine Zahl von 0 bis 10.
```

1.2 Schleifen

Computer sind besonders gut geeignet, um repetitive Aufgaben zu lösen. Solche Aufgaben werden durch Schleifen definiert. Julia bietet zwei Arten von Schleifen: `for`- und `while`-Schleife.

Die `while`-Schleife hat eine gewisse Ähnlichkeit zur `if`-Bedingung. Eine Aufgabe in einer solchen Schleife wird immer wieder ausgeführt, solange eine *Bedingung* erfüllt ist.

Im folgenden Beispiel wird jede Zahl von 0 bis 10 ausgegeben.

```
function zähle_bis(n)
    i = 1
    while i <= n
        println("Bin bei Zahl $i")
        i += 1
    end
end
zähle_bis(10)
```

```
Bin bei Zahl 1
Bin bei Zahl 2
Bin bei Zahl 3
Bin bei Zahl 4
Bin bei Zahl 5
Bin bei Zahl 6
Bin bei Zahl 7
Bin bei Zahl 8
Bin bei Zahl 9
Bin bei Zahl 10
```

Der Code wird folgendermaßen durchschritten:

1. Initialisiere `i` mit `1`
2. Falls die Bedingung `i <= 10` erfüllt ist, gehe zu (3), ansonsten gehe zu (6)
3. Ausgabe von `"Bin bei Zahl $i"`
4. Erhöhe `i` um `1`
5. Gehe zurück zu (2)
6. Ende

Dieser Ablauf wird Schleife genannt, weil (5) wieder zurück zu (2) springt. Bei diesem Code ist Vorsicht geboten: Vergisst man die Zeile `i += 1`, ergibt die Bedingung `i <= 10` nie `false`. Das bedeutet, die Schleife läuft endlos (Endlosschleife). In der REPL oder in VS Code kann eine Endlosschleife durch die Tastenkombination `Ctrl+C` abgebrochen werden.

Beim obigen Beispiel handelt es sich um eine Zählschleife. Dieses Konstrukt ist so häufig, dass es einen eigenen Schleifentyp dafür gibt: Die `for`-Schleife. Der Code in der `for`-Schleife wird für jedes Element in einer ihr übergebenen Menge ausgeführt. Das vorherige Beispiel vereinfacht sich daher zu:

```
function zähle_bis(n)
    for i in 1:n
        println("Bin bei Zahl $i")
    end
end
zähle_bis(10)
```

```
Bin bei Zahl 1
Bin bei Zahl 2
Bin bei Zahl 3
Bin bei Zahl 4
Bin bei Zahl 5
Bin bei Zahl 6
Bin bei Zahl 7
Bin bei Zahl 8
Bin bei Zahl 9
Bin bei Zahl 10
```

Anstelle von `1:n` kann jede Art von Liste angegeben werden.

```
for buchstabe in ["A", "B", "C", "D"]
    println("Bin bei Buchstabe $buchstabe")
end
```

```
Bin bei Buchstabe A
Bin bei Buchstabe B
Bin bei Buchstabe C
Bin bei Buchstabe D
```

1.2.1 break und continue

Eine Schleifeniteration kann jederzeit durch den Aufruf von `break` abgebrochen und den Aufruf von `continue` übersprungen werden.

Im folgenden Beispiel werden von einem `String` nur die Vokale gesammelt. Falls ein `!` gefunden wird, werden die gesammelten Vokale ausgegeben und die Schleife beendet.

```
function sammle_vokale(text)
    vokale_sammlung = ""
    for buchstabe in text
        if buchstabe == '!'
            println(vokale_sammlung)
            break  # Beende die Schleife
        end
        if !(buchstabe in ('a','e','i','o','u'))
            continue  # Zurück zum Beginn der Schleife
        end
        vokale_sammlung *= buchstabe
    end
end
sammle_vokale("Finde die Vokale in diesem Satz. Gehe aber nicht weiter als bis zum !. Das sollst du nicht mehr ansehen")
```

```
ieieoaeiieeaeeaeieieaiu
```

`break` wird oft auch in Kombination mit `while`-Schleifen verwendet, die endlos laufen. Wenn beispielsweise in der Schleife auf eine Dateneingabe gewartet wird oder die Abbruchbedingung sehr komplex ist. In diesem Fall kann eine Endlosschleife absichtlich durch Verwenden der Bedingung `true` konstruiert werden.

1.2.2 Hilfreiche Funktionen bei Schleifen

Häufig kommt es vor, dass in einer `for`-Schleife Wert und Index eines Elements benötigt werden. Dafür gibt es die praktische Funktion `enumerate`, die auf ein iterierbares Objekt angewendet, den Index und den Wert zurück gibt. In der `for`-Schleife können die beiden Rückgabewerte durch Angabe von zwei Variablennamen in runden Klammern zugewiesen werden:

```
for (index,buchstabe) in enumerate(["A", "B", "C", "D"])
    println("Buchstabe $buchstabe steht an der Stelle $index")
end
```

```
Buchstabe A steht an der Stelle 1
Buchstabe B steht an der Stelle 2
Buchstabe C steht an der Stelle 3
Buchstabe D steht an der Stelle 4
```

1.2.3 Grundregeln für effizienten Code

Bei den folgenden Aufgaben ist es wichtig, möglichst effizienten Code zu schreiben. Optimierung ist ein sehr komplexes Thema, es gibt in Julia aber ein paar grundlegende Punkte, die eine Effizienzsteigerung bringen.

Um eine Auskunft über die Laufzeit eines Codes zu erhalten (Benchmark-Test), biete Julia das `@time`-Macro.

Die folgende Funktion berechnet `n`-mal die Inverse einer Matrix `A` und ist nur als Beispielfunktion für die Verwendung von `@time` gedacht.

```
function time_dummy(n::Int)
    A = [3 1 0
         0 1 0
         3 3 3]
    for i in 1:n
        inv(A)
    end
end
```

```
time_dummy (generic function with 1 method)
```

```
julia> @time time_dummy(100000)
  0.154855 seconds (536.84 k allocations: 208.363 MiB, 19.35% gc time, 29.01% compilation time)

julia> @time time_dummy(100000)
  0.080718 seconds (500.00 k allocations: 205.994 MiB, 17.29% gc time)

julia> @time time_dummy(100000)
  0.081211 seconds (500.00 k allocations: 205.994 MiB, 16.47% gc time)
```

Das `@time`-Macro gibt die erforderliche Zeit und Angaben über das Speichermanagement aus. Der erste Aufruf der neu geschriebenen Funktion `time_dummy` braucht wesentlich länger als die nächsten beiden. Das liegt daran, dass Julia beim ersten Aufruf die Funktion für die angegebenen Argumente kompiliert.

Dieser Vorgang ist sehr wichtig, um eine performante Ausführung zu ermöglichen, und wird stark durch die sogenannte *Typenstabilität* beeinflusst.

Typenstabilität Stabilität bedeutet hier, dass sich der Datentyp einer Variable mit der Laufzeit nicht ändert. Eine mögliche Fehlerquelle dafür ist im folgendem Beispiel zu sehen[1].

```
function summe_sin_A(n::Int)
    ∑ = 0
    for i in 1:n
        ∑ += sin(n)
    end
    return ∑
end

function summe_sin_B(n::Int)
    ∑ = 0.0
    for i in 1:n
        ∑ += sin(n)
    end
    return ∑
end
```

Der einzige Unterschied der Funktionen A und B befindet sich in der ersten Zeile bei der Definition von `∑` (`\sum`+`TAB`-Taste). Im Fall von A wird mit `Int` im Fall von B mit `Float64` initialisiert.

Um die Funktionen zu benchmarken, werden beide in Schleifen ausgeführt.

```
function benchmark_summe_A()
    for _ in 1:100
        summe_sin_A(100000)
    end
end

function benchmark_summe_B()
    for _ in 1:100
        summe_sin_B(100000)
    end
end
```

Die Laufzeit der beiden Fälle ergibt sich zu:

```
julia> benchmark_summe_A(); benchmark_summe_B(); # Erster Aufruf zum Kompilieren

julia> @time benchmark_summe_A()
  0.005158 seconds
```

[1] http://www.johnmyleswhite.com/notebook/2013/12/06/writing-type-stable-code-in-julia/

```
julia> @time benchmark_summe_B()
  0.000000 seconds
```

Der große Unterschied zwischen den beiden Funktionen resultiert aus dem Rückgabewert der Funktion `sin`:

```
julia> typeof(sin(3))
Float64
```

`sin` gibt `Float64` zurück. Das bedeutet bei Fall A muss `Σ` von `Int` in `Float64` konvertiert werden, was immens viel Zeit benötigt.

Ein weiterer häufig gemachter Fehler tritt bei Funktionen auf, die für gleiche Datentypen der Argumente verschiedene Datentypen der Ergebnisse zurückgeben. Beispielsweise gibt die folgende Definition der Rampen-Funktion inkonsistente Datentypen zurück:

```
function ramp(x)
    if x < 0
        return 0
    else
        return x
    end
end
```

```
julia> ramp(3.0)
3.0

julia> ramp(-3.0)
0

julia> ramp(3)
3

julia> ramp(-3)
0
```

Eine Möglichkeit wäre, die Funktion für die Typen `Float64` und `Int` zu spezialisieren oder besser die `zero`-Funktion zu verwenden:

```
function ramp(x)
    if x < 0
        return zero(x)
    else
        return x
    end
end
```

```
julia> ramp(3.0)
3.0

julia> ramp(-3.0)
0.0

julia> ramp(3)
3

julia> ramp(-3)
0
```

Vermeiden von globalen Variablen Eine globale Variable ist beispielsweise `x` im folgenden Code:

```
x = "Ich bin global!"

function test_global()
    println(x)
end
test_global()
```

```
Ich bin global!
```

`x` ist innerhalb der Funktion `test_global` verwendbar, obwohl die Variable außerhalb definiert wurde. Der Nachteil von globalen Variablen, aus Sicht der Optimierung, ist, dass sie von mehreren Stellen aus verändert werden

können. Das führt dazu, dass Julia die Funktionen, die die Variablen verwenden, wieder nicht exakt für den Typ der globalen Variable spezialisieren kann, da sich dieser Ändern kann:

```
x = 100
test_global()
```

```
100
```

`test_global` gibt jetzt den neuen Wert von `x` aus. In diesem Fall hat sich nicht nur der Wert, sondern auch der Typ von `String` zu `Int` geändert.

In manchen Fällen ist es sinnvoll, globale Variablen zu verwenden, dann sollten diese mit dem Keyword `const` als Konstanten definiert werden (Konstanten werden üblicherweise in Blockbuchstaben geschrieben).

```
const GRAV_BESCHL = 9.81 # m/s²
function Kraft(masse) # masse in kg
    masse*GRAV_BESCHL
end
Kraft(8.0)
```

```
78.48
```

Eine Konstante darf sich nicht ändern; beim Versuch eine Konstante zu ändern, gibt Julia einen Error zurück. Damit ist der Code wieder typenstabil.

```
GRAV_BESCHL = "Neuer Wert"
```

```
Error: invalid redefinition of constant GRAV_BESCHL
```

1.3 Aufgaben

1.3.1 Gerade oder Ungerade

Schreibe eine Funktion, die „Gerade“ oder „Ungerade“ für eine Zahl mit `println` ausgibt.

Beispiele:

```
julia> gerade_ungerade(11)
Ungerade

julia> gerade_ungerade(218)
Gerade
```

1.3.2 Skript Flächeninhalt

Es soll ein **Skript** zur Berechnung von Flächeninhalt und Umfang eines Kreises erstellt werden, das BenutzerInnen erlaubt, mehrmals Eingaben zu machen. Die Ausgabewerte sollen auf zwei Stellen gerundet werden. Das Skript soll erst abgebrochen werden, wenn eine leere Eingabe gemacht wird. Wird eine falsche Eingabe gemacht, muss „Falsche Eingabe“ ausgegeben werden.

Um eine Eingabe von der Tastatur einzulesen, kann die folgende Funktion verwendet werden:

```
"""
    wert_einlesen(text)

Lies eine Texteingabe vom Benutzer ein, gib als Aufforderung `text` aus und gib den eingegebenen Wert
als String zurück.
"""
function wert_einlesen(text::AbstractString)
    # Aufforderung ausgeben
    print(text)
    print(": ")
    # Eine Zeile von der Benutzereingabe einlesen
    return readline()
end
```

Die Ausgabe nach Ausführung des Skripts mit `julia kreis.jl` muss exakt mit dem folgenden Text übereinstimmen.

```
Radius eingeben: 3
A = 28.27
U = 18.85

Radius eingeben: abc
Falsche Eingabe

Radius eingeben: 1.2
A = 4.52
U = 7.54

Radius eingeben:
```

Für die Kontrolle mit dem Kontrollprogramm muss die allgemeine Form der Eingabe verwendet werden. Zusätzlich muss die Schleife für die wiederholte Ausführung über eine Funktion gestartet werden. Die Funktion muss wie im folgenden Beispiel am Ende des Kontrollblocks stehen. Wichtig für die Kontrolle ist, dass alle Ausgaben exakt mit dem Beispiel oben übereinstimmen (Achtung: Nach der Ausgabe befindet sich eine Leerzeile).

```
@Aufgabe "11.3.2" begin
    # ...
    # Platz für Zusatzfunktionen die eventuell verwendet werden
    # ...
    function run()
        # ...
        # Hauptfunktion die die Schleife beinhaltet
        # ...
        while
            # ...
        end
        # ...
    end
end
```

1.3.3 Berechne π

Die Zahl π kann über eine so genannte Monte-Carlo-Simulation angenähert werden. Dabei werden zufällig Punkte in einem Quadrat mit bekannter Seitenlänge platziert. Durch Einschreiben eines Kreise in das Quadrat und Ermitteln der Anzahl an Punkten, die im Kreis gelandet sind, kann die Fläche des Kreises relativ zur Fläche des Quadrates ermittelt werden. π wird dann einfach über die Kreisflächenformel berechnet.

Schreibe eine Funktion, die π mittels Monte-Carlo-Simulation approximiert. Das einziges Argument der Funktion ist die Anzahl an zufällig ermittelten Punkte. Um den Rechenaufwand zu reduzieren, soll nur ein Vierteilkreis in einem Quadrat berücksichtigt werden. Die ermittelte relative Fläche muss dann natürlich noch mit dem Faktor 4 vergrößert werden.

Beispiele (mit steigender Anzahl von Punkten steigt auch die Genauigkeit):

```
julia> π_mc(10)
2.8

julia> π_mc(100)
3.08

julia> π_mc(1_000_000)
3.143648

julia> π_mc(1_000_000_000)
3.141646796
```

In Julia kann zur besseren Lesbarkeit ein `_` in Zahlen eingefügt werden. Im vorherigen Beispiel dient `_` als Tausendertrennzeichen, hat aber keinen Einfluss auf die Zahl selbst (`100 == 1_00`).

1.3.4 Numerisch Integrieren

Schreibe eine Funktion, die das bestimmte Integral einer beliebigen 1D-Funktion, definiert durch x- und y-Werte, berechnet. Verwende dazu die Trapezregel[2].

[2] https://de.wikipedia.org/wiki/Trapezregel

Beispiele:

```
julia> x = collect(0:0.0001:1);

julia> ∫(x,sin.(x.^2))
0.3102683026238847

julia> ∫(x,cos.(x.^2))
0.9045242364978205

julia> ∫(x,x)
0.4999999999999999
```

Je feiner die Auflösung, desto genauer wird das Ergebnis.

```
julia> x = collect(0:0.1:2π);

julia> ∫(x,sin.(x))
0.003455020910590271

julia> x = collect(0:0.01:2π);

julia> ∫(x,sin.(x))
5.0730443500457005e-6

julia> x = collect(0:0.0001:2π);

julia> ∫(x,sin.(x))
3.6386392119709672e-9
```

1.3.5 Wie hoch ist die Wahrscheinlichkeit

Die Monte-Carlo-Simulation kann auch dazu verwendet werden, um Wahrscheinlichkeiten zu approximieren. Schreibe zwei Funktionen, welche die Wahrscheinlichkeit abschätzen, dass

1. mit einem idealen Würfel (1–6) 3× hintereinander 4 gewürfelt wird und
2. mit einem idealen Würfel (1–6) bei 3 Würfen mindestens ein 4er gewürfelt wird.

Die Idee hinter der Simulation ist, die Würfe mit einem Zufallszahlengenerator `rand(1:6)` zu simulieren und anschließend die positiven Ergebnisse in Relation zu den gesamten Würfen zu setzten.

Wie schon bei der Berechnung von π soll das einzige Argument bei den Funktionen die Anzahl an durchzuführenden Simulationen sein (eine Berechnung entspricht dreimal würfeln). Die Nummern für die Kontrolle sind `11.3.5.1` und `11.3.5.2`.

Beispiele:

```
julia> P_est_1(10)
0.0

julia> P_est_1(100)
0.01

julia> P_est_1(1_000_000)
0.004632

julia> P_est_1(1_000_000_000)
0.004629473

julia> P_est_2(10)
0.4

julia> P_est_2(100)
0.34

julia> P_est_2(1_000_000)
0.420631

julia> P_est_2(1_000_000_000)
0.421329607
```

Die exakten Ergebnisse sind:

```
julia> 1/6^3 # Fall 1
0.004629629629629629

julia> 1/6+5/6*1/6+5/6*5/6*1/6 # Fall 2
0.42129629629629634
```

1.3.6 Split

Im Abschnitt zu Strings wurde die Funktion `split` vorgestellt, mit der Strings durch Trennen bei einem Zeichen, in ein String-Array umgewandelt werden können. Bei dieser Aufgabe soll die Funktion nachprogrammiert werden, natürlich ohne die Verwendung von `split`. Das Trennzeichen muss vom Typ `char` sein und im Gegensatz zur `split` Funktion darf das resultierende Array keine leeren Strings `""` enthalten. Achtung, auch die Sonderfälle

- kein Trennzeichen
- Trennzeichen am Anfang
- Trennzeichen am Ende und
- mehrere Trennzeichen nacheinander

müssen behandelt werden.

Beispiele:

```
julia> my_split("Das ist ein String",' ')
4-element Vector{String}:
 "Das"
 "ist"
 "ein"
 "String"

julia> my_split(".Das.ist.ein..String.",'.')
4-element Vector{String}:
 "Das"
 "ist"
 "ein"
 "String"
```

1.3.7 Wurzelziehen

Nicht immer war Wurzelziehen so komfortabel wie heute. Mit der Unterstützung eines Computers oder Taschenrechners gelingt das ganz einfach mit der Funktion `sqrt`. In der Literatur findet man allerdings einige Verfahren, mit welchen „händisches“ Wurzelziehen möglich ist. Eines dieser Verfahren wird babylonisches Wurzelziehen[3] genannt.

Zur Berechnung der Quadratwurzel x von a wird zuerst eine Zahl x_0 als Kandidat gewählt (z.B. $x_0 = \frac{a}{2}$). Anstelle von $x = \sqrt{a}$ wird dann die Gleichung mit dem Kandidaten $x_0 \cdot x_0 \stackrel{!}{=} a$ gelöst. Wurde zufällig $x_0 = x$ gewählt, ist der Algorithmus beendet. Wahrscheinlicher ist jedoch, dass $x_0 \cdot x_0 \neq a$. Je nach der Wahl von x_0 ist x_0^2 jetzt größer oder kleiner als a, was auch bedeutet, dass x_0 jetzt entsprechend größer oder kleiner als x ist. Ändern wir die Gleichung für a zu $x_0 \cdot b = a$ können wir durch Umformen eine zweite Zahl b ermitteln, für die gilt

$$\begin{cases} b < x, & \text{wenn } x_0 > x \\ b > x, & \text{wenn } x_0 < x \end{cases}$$

Damit wurde eine obere und eine untere Schranke für x bestimmt, wobei jeder Wert innerhalb dieser Schranke eine bessere Annäherung and die Quadratwurzel ist. Der nächste Wert x_1 wird dann als Mittelwert von x_0 und b ermittelt. Allgemein ergibt sich daraus der Ausdruck

$$x_{n+1} = \frac{x_n + \frac{a}{x_n}}{2}.$$

[3] https://de.wikipedia.org/wiki/Heron-Verfahren

Für eine ausreichend hohe Zahl von n kann damit die Quadratwurzel von a ermittelt werden. Es soll eine Funktion definiert werden, die mit dem beschriebenen Verfahren die Quadratwurzel einer beliebigen Zahl mit einem absoluten Fehler von 10^{-5} berechnet. Ein solches Abbruchkriterium wird auch Konvergenzkriterium genannt.

Beispiele:

```
julia> bsqrt(9)
3.0000000000393214

julia> bsqrt(103.12)
10.154801819992677
```

1.3.8 Dictionaries zusammenfügen

Schreibe eine Funktion `dict_merge`, die zwei Dicts vom gleichen Typ als Argumente nimmt, diese beiden Dictionaries zusammenführt (merged) und als Return Value zurückgibt. Sind Keys in beiden `Dicts` vorhanden, sollen die Werte aus dem ersten mit den Werten aus dem zweiten `Dict` überschrieben werden.

```
julia> d1 = Dict("Knopparp"=>71.99,"Söderhamn"=>314.1)
Dict{String, Float64} with 2 entries:
  "Söderhamn" => 314.1
  "Knopparp"  => 71.99

julia> d2 = Dict("Landskrona"=>494.1,"Ektorp"=>359.1)
Dict{String, Float64} with 2 entries:
  "Ektorp"     => 359.1
  "Landskrona" => 494.1

julia> dict_merge(d1, d2)
Dict{String, Float64} with 4 entries:
  "Ektorp"     => 359.1
  "Söderhamn"  => 314.1
  "Landskrona" => 494.1
  "Knopparp"   => 71.99
```