Einführung in Julia

Algorithmen und Datenstrukturen

Magnus Bender

20. April 2023

UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR INFORMATIONSSYSTEME

Motivation

Julia im Vergleich zu Java

```
function summe(A)
  s = 0
  for i = 1:length(A)
    s = s + A[i]
  end
  return s
end


A = [1, 2, 3, 4, 5]
s = summe(A)
println("summe($A) = $s")
```

```
import java.util.Arrays;

public class Summe {
  public int summe(int[] A) {
    int s = 0;
    for (int i = 0; i < A.length; i++) {
      s = s + A[i];
    }
    return s;
  }

  public static void main(String args[]) {
    int[] A = {1, 2, 3, 4, 5};
    int s = new Summe().summe(A);
    System.out.println("summe("+
      Arrays.toString(A) +") = " + s);
  }
}
```

138 und 337 Zeichen

Einführung

Was ist Julia?

- Julia ist eine moderne und effiziente Programmiersprache
- Verschiedenste Anwendungsgebiete
  - Web Applikationen
  - Data Science
  - Wissenschaftliches Rechnen

Einführung

Ziel der Vorlesung

- Grundlegende Syntax von Julia
- Ausführen der Programme auf den Folien
- Lösen von Übungsaufgaben
- Fokus auf Verstehen der Algorithmen
- Nachvollziehen ohne Fokus auf Dinge wie Typen etc.

Programmieraufgaben auf den Übungszetteln und die Mid-Term-Programmieraufgabe sind in Julia zu lösen. (Nutzung des VPL)

Software

Wir nutzen VS Code mit der Julia-Extension

Julia

https://julialang.org/downloads/

VS Code

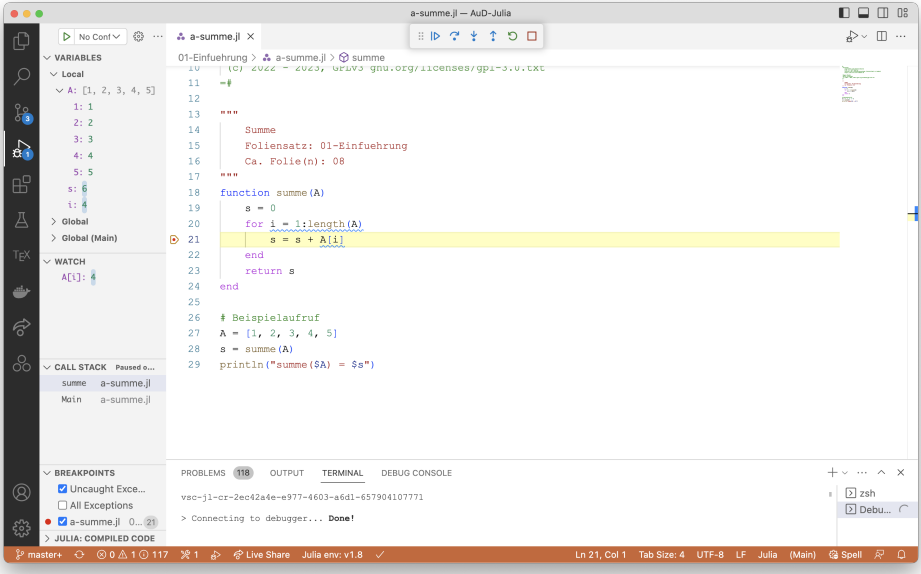
https://code.visualstudio.com/download

Extension Julia

https://code.visualstudio.com/docs/languages/julia

- Ausführen von Julia-Programmen
- Debugging von Julia-Programmen
  - Breakpoints
  - Variablenbelegungen anzeigen

VS Code mit Julia



Informationsquellen

Webseite

https://julialang.org/

Dokumentation

https://docs.julialang.org/en/v1/

Code von den Folien

https://www.ifis.uni-luebeck.de/~bender/lehre/ss23/aud/julia-code.zip

Interaktives Terminal

Read-Eval-Print Loop (REPL)

```
magnus@laptop$ julia

  julia> a = 1 + 1
    2
  julia> b = 3 * a
    6
  julia> exp(b)
    403.4287934927351
  julia> exit()

magnus@laptop$
```

- Ein Ausdruck in einer Zeile oder mit `;` trennen
- Einrückung wird ignoriert

Übersicht

Vergleiche

```
==, !=
>, >=, <, <=
```

Operatoren

```
+, -, *, /, %, ^
&&, ||, !
```

Bedingungen

```
if i < 10
    ...
elseif i < 20
    ...
else
    ...
end
```

Schleifen

```
for i = 1:n
    ...
end

while i < 12
    ...
end
```

Zuweisung

```
x = 5
x = x + 5
```

Kommentare

- Alles nach #
- Alles zwischen #= und =#
- Alles zwischen " und "

Namen

- Case sensitive
- Unicode, mathematische Symbole wie π, $\in$

Datentypen

Elementar

Nothing `nothing` analog zu `null` bzw. `void*`

Bool `true` und `false`

Int, Int8, Int32 1, 100, 3236

Float32, Float64 1.0, 3.141593, 2.718282, *NaN*, *Inf*, $-Inf$

Char `'H'`, `'c'`

String `"Hello"`, `"How are you?"`, `"123"`

Missing `missing` stellt einen fehlenden Wert dar, z.B. in der Statistik

Any stellt ein Element *beliebigen* Typs dar

Datentypen

Zusammengesetzt

Array Multidimensionale dynamische Arrays, z.B. Änderungen der Größe möglich

Vector Eindimensionales Array

Matrix Zweidimensionales Array

Zugriffe

- 1-Indexiert
- Zugriffe mit `matrix[1][1]` oder auch `matrix[1,1]`
- Intervalle mit `:` wählbar, z.B. `matrix[2,3:5]`
- Zeilen oder Spalten vollständig, z.B. `matrix[1,:]` oder `matrix[:,1]`

Datentypen

Eigene Typen, Objekte

```
struct Studi          # Typendefinition
  name :: String
  jahrgang :: Int
  matrikelnummer
end

s = Studi("Otto", 1999, 111222)

using Dates           # year(), now()

function print_studi(s :: Studi)
  println("Name: " * s.name)
  println(string("Alter: ", year(now()) - s.jahrgang))
  println("Matrikelnummer: $(s.matrikelnummer)")
end

print_studi(s)
```

Datentypen

Beispiele

```
julia> [11 12; 13 14]
  2×2 Matrix{Int64}:
    11  12
    13  14

julia> ["BMW" 12; "Audi" 11]
  2×2 Matrix{Any}:
    "BMW"   12
    "Audi"  11

julia> [["BMW", 12], ["Audi", 11]]
  2-element Vector{Vector{Any}}:
    ["BMW", 12]
    ["Audi", 11]
```

Funktionen

Benutzen

- Funktionsnamen sind selbst Variablen

```
julia> exp(1)
  2.718281828459045
julia> exp
  exp (generic function with 14 methods)
```

- Funktionen können als Parameter übergeben werden

```
julia> map(exp, [1, 2])
  2-element Vector{Float64}:
    2.718281828459045
    7.38905609893065
```

- Funktionen können überschrieben werden

```
julia> length = 2
julia> length([1,2,3,4])
  MethodError: objects of type Int64 are not callable
```

Funktionen

Erstellen

```
function function_name(argument_1, argument_2 = default, ...)
  # Funktionskörper

  return value_1, value_2, ...
end
```

- Schlüsselwort **`function`** für Funktionsdefinition
- Argumente sind mit Komma getrennte Liste der Form
  `name [= ausdruck]`
- Rückgabewert(e) mittels **`return`** oder Wert des letzten Ausdrucks

Funktionen

Konventionen

Seiteneffekte

Funktionen, die Seiteneffekte auf übergebene Argumente auswirken, werden üblicherweise mit einem `!` am Ende bezeichnet.

```
julia> A = []
julia> push!(A, 1)
  1-element Vector{Any}:
    1
```

Nützliche Funktionen

- `length(A)` Bestimmt die Anzahl Elemente eines Arrays
- `isempty(A)` Prüft, ob ein Array leer ist
- `sum(A), maximum(A), minimum(A)` Summe, Maximum, Minimum der Elemente eines Arrays

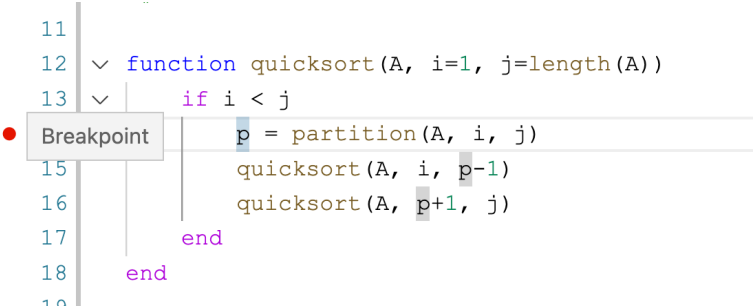
- `push!(A, value)` Fügt einen Wert hinten an ein Array an
- `pop!(A)` Entfernt den letzten Wert eines Arrays und gibt ihn zurück
- `append!(A, values)` Fügt mehre Elemente (z.B. aus einem zweiten Array) an ein Array an
- `insert!(A, index, value)` Fügt einen Wert an einen gegebenen Index in ein Array ein
- `deleteat!(A, index)` Löscht ein Element an einem Index aus einem Array

Debugging

- Julia-Programm während der Ausführung beobachten
  - Ausführung anhalten
  - Variablen auslesen
  - Kontrollfluss überwachen

Debugging
in VS Code

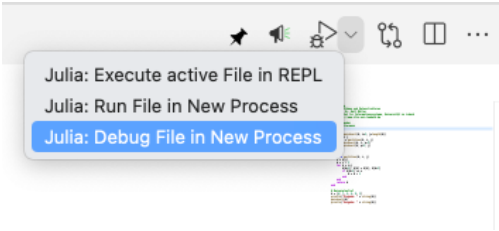
1. Breakpoint setzen



2. Debugger starten
3. Variablen auslesen
4. Kontrollfluss überwachen

Debugging
in VS Code

1. Breakpoint setzen
2. Debugger starten

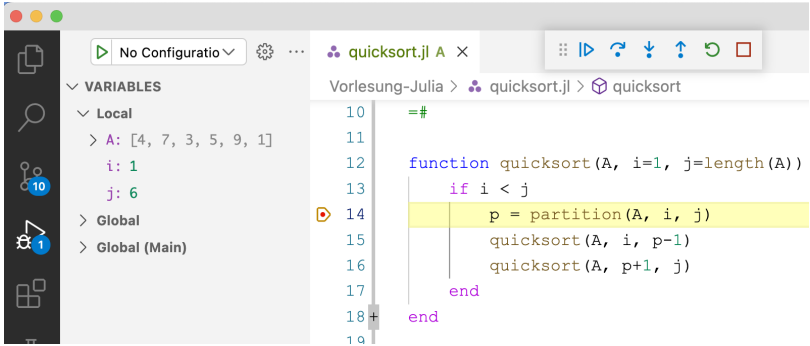


3. Variablen auslesen
4. Kontrollfluss überwachen

Debugging

in VS Code

1. Breakpoint setzen
2. Debugger starten
3. Variablen auslesen



4. Kontrollfluss überwachen

Debugging
in VS Code

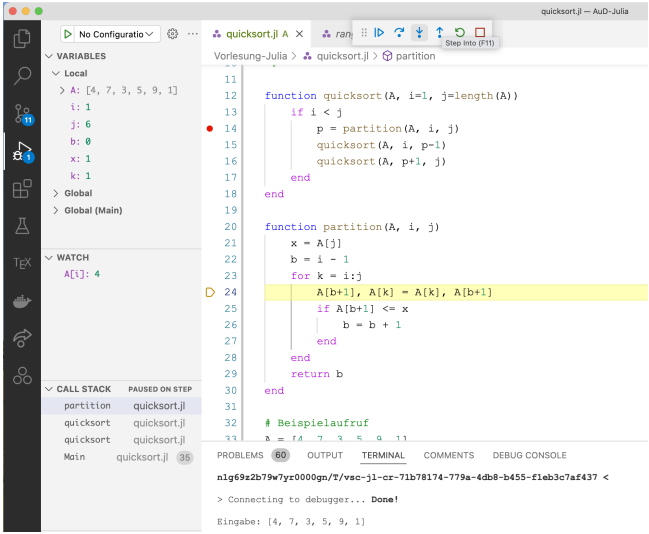
1. Breakpoint setzen
2. Debugger starten
3. Variablen auslesen
4. Kontrollfluss überwachen

Continue Bis zum Ende/ nächsten Breakpoint fortsetzen
Step Over Ein Schritt/ einen Block weiter gehen
Step Into In den Block/ die Funktion reingehen
Step Out Zum Ende des Blocks/ der Funktion gehen
Restart Debugger neu starten
Stop Debugger stoppen

Debugging

in VS Code

4. Kontrollfluss überwachen



Live-Demo

`summen-demo.jl`

Danke für die Aufmerksamkeit!

Magnus Bender

m.bender@uni-luebeck.de

Fragen?!