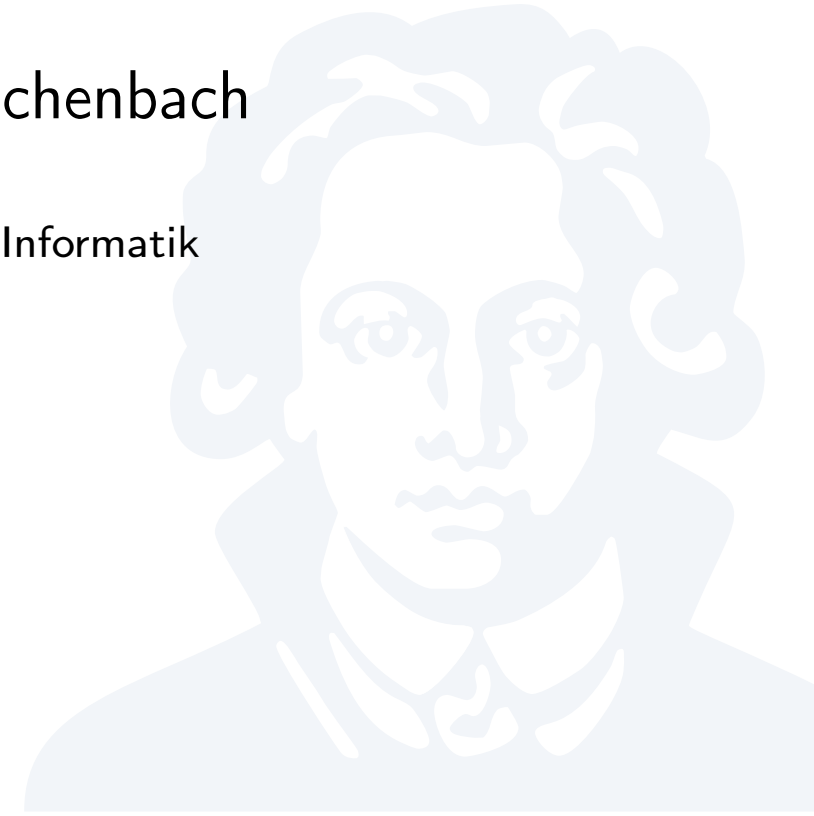
Einführung in die Programmierung mit *Go*

Teil 1: Grundlegende Konzepte

Prof. Dr. Christoph Reichenbach

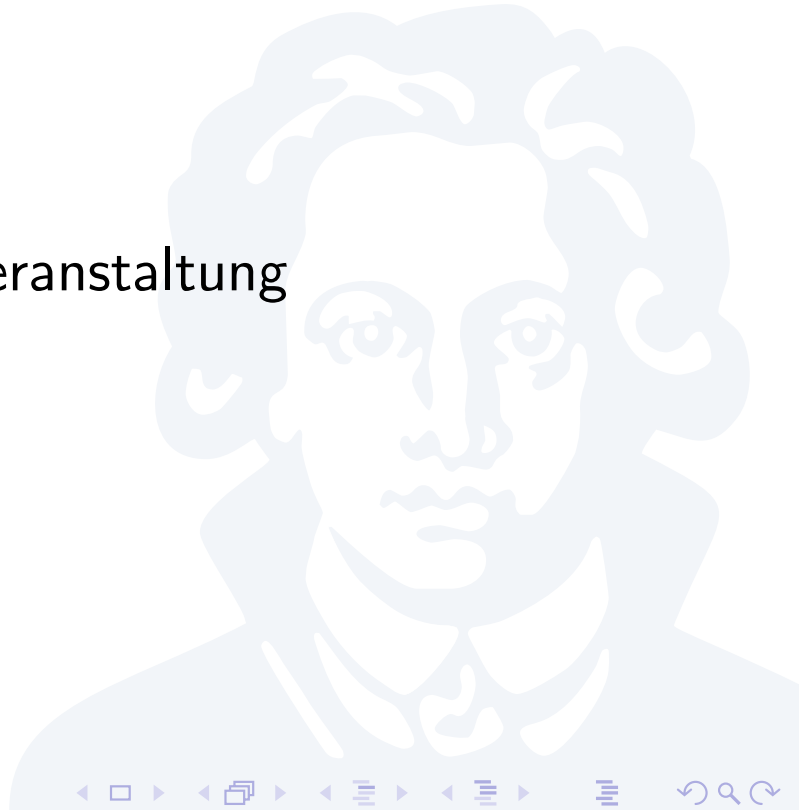
Fachbereich 12 / Institut für Informatik

8. März 2015



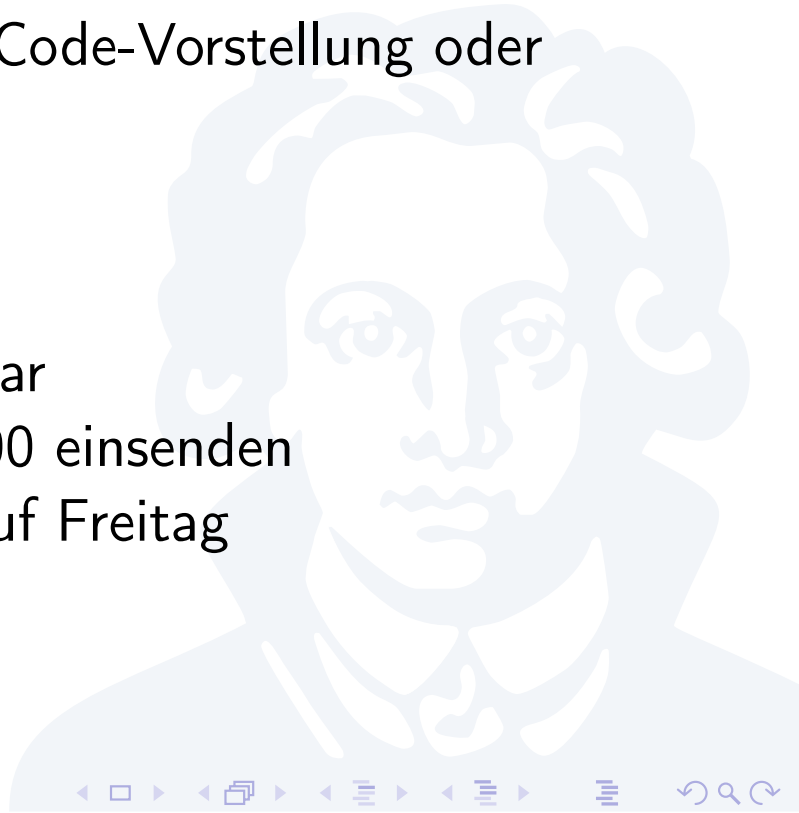
Übersicht der Veranstaltung

- Sonderveranstaltung: 5 Tage
  - 1h Vorlesung
  - 1h Übung
- Projekt
  - Notwendig zum Bestehen der Veranstaltung

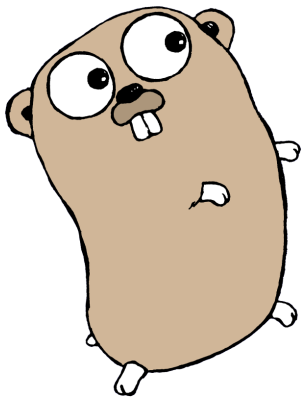


Projekte

- Selbstständig oder in Zweier-Team in go entwickelt
- Vorstellung:
  - **Datum**: Freitag, der 27 März, ab 09:00
  - **Ort**: in Seminarraum 11
  - 15 Minuten Demonstration mit Code-Vorstellung oder Vortrag
  - Einsenden des Codes
- Projektinhalt:
  - Vorschläge ab Mittwoch verfügbar
  - Präferenzen bis Donnerstag 18:00 einsenden
  - Zuweisung erfolgt Donnerstag auf Freitag
  - Eigene Ideen willkommen



Steckbrief: Go



Go

Version:	**1.4.2**
Datum:	**2015-02-18**
Quelle:	**http://golang.org**

Laufzeitsystem

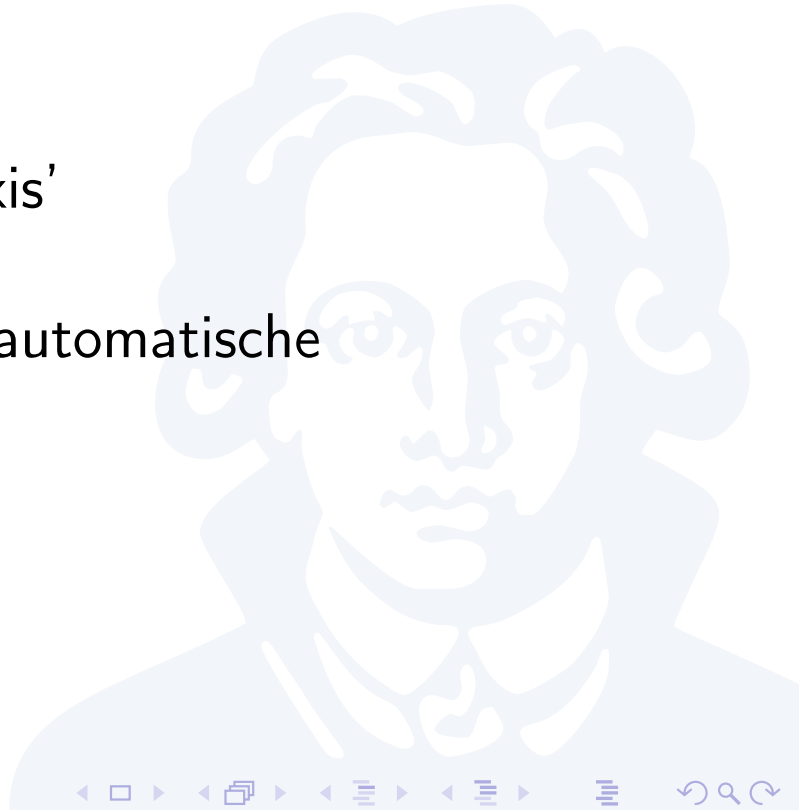
Ausführungsmodus:	**Übersetzt**
Speicherverwaltung:	**Automatisch**

Sprache

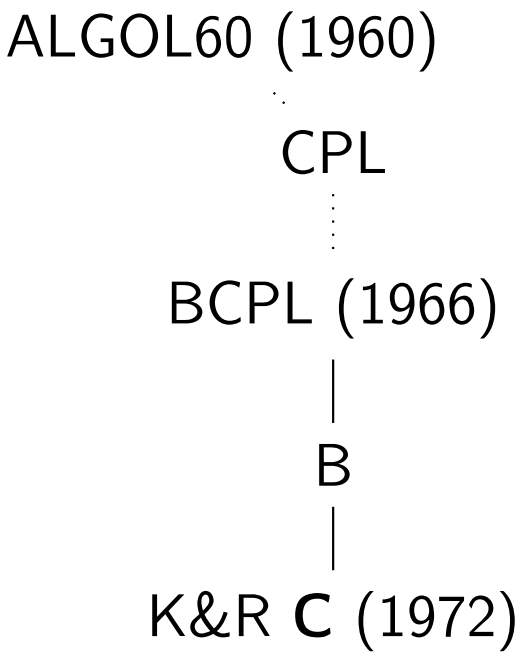
Sprachtyp:	**Imperativ**
Starke Typisierung:	**größtenteils** (statisch und dynamisch)
Sprachkonstrukte:	CSP-Kanäle, Threads, Funktionen-als-Werte, Reflektion

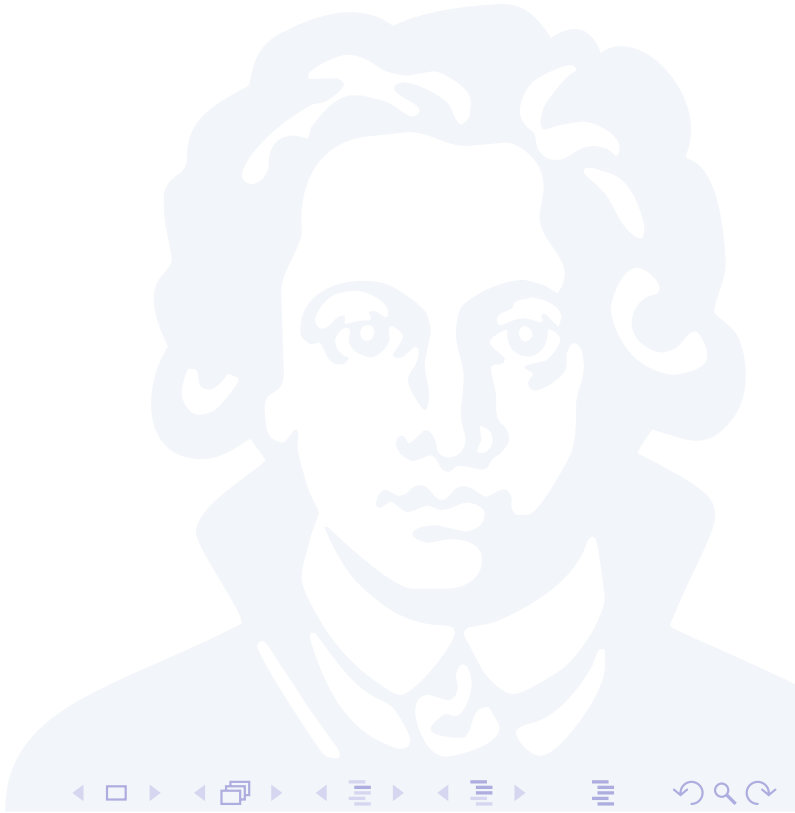
Hintergründe zu *Go*

- Sprache zur Systemprogrammierung
  - (Relativ) nah an Hardware
  - Kontrolle wichtiger als Bequemlichkeit
- Praktische Sprache
  - Schnell zu übersetzen
  - Erzwingt ‘gute Programmierpraxis’
  - Beinhaltet Software-Werkzeuge
  - Vermeidet Speicherfehler durch automatische Speicherverwaltung
- Gegenentwurf zu C++



Go und seine Verwandtschaft





Go und seine Verwandtschaft

ALGOL60 (1960)

CPL

BCPL (1966)

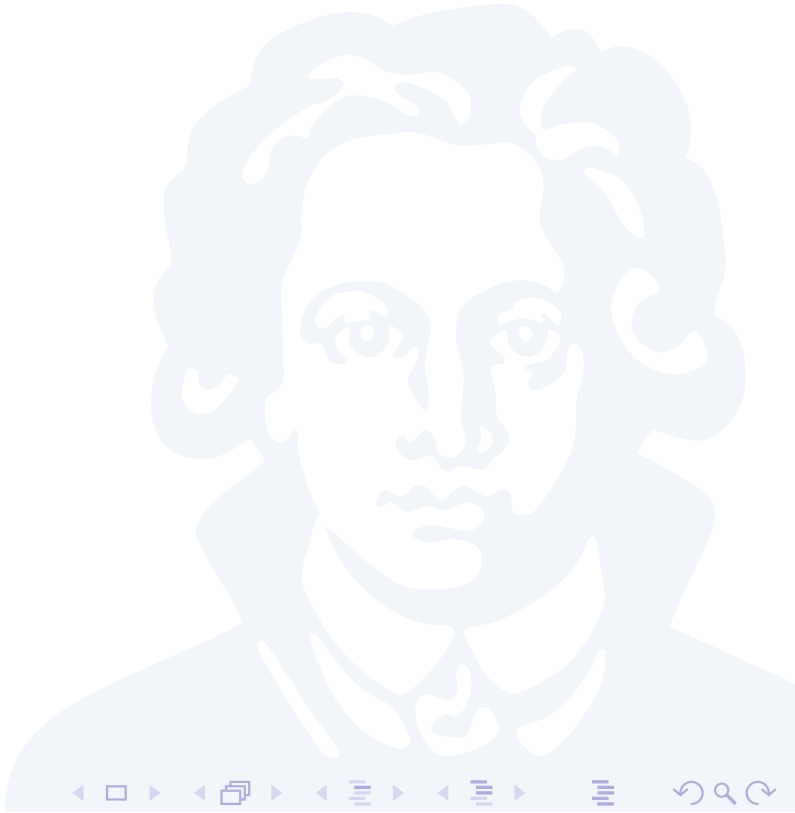
B

K&R **C** (1972)

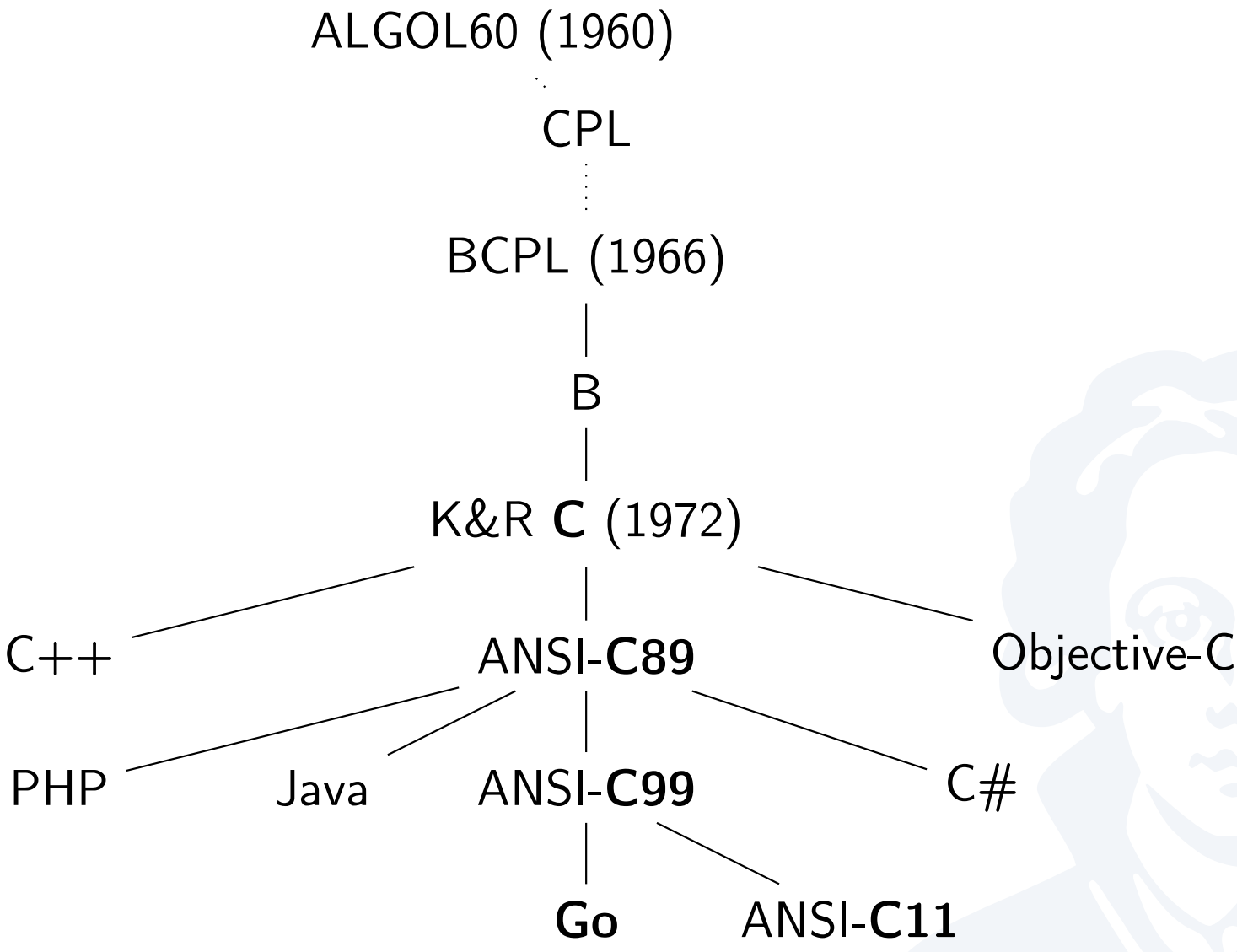
ANSI-**C89**

ANSI-**C99**

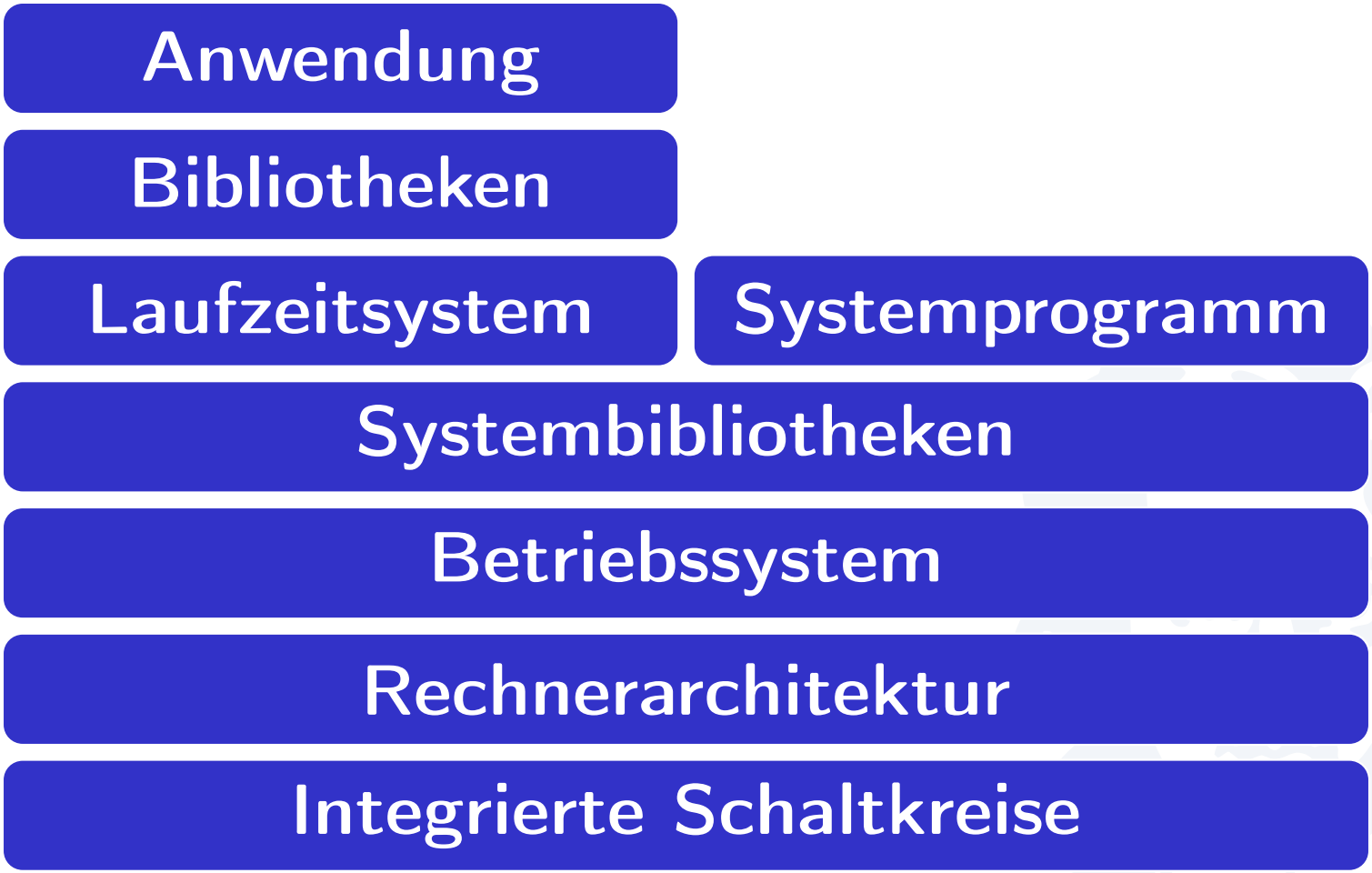
Go



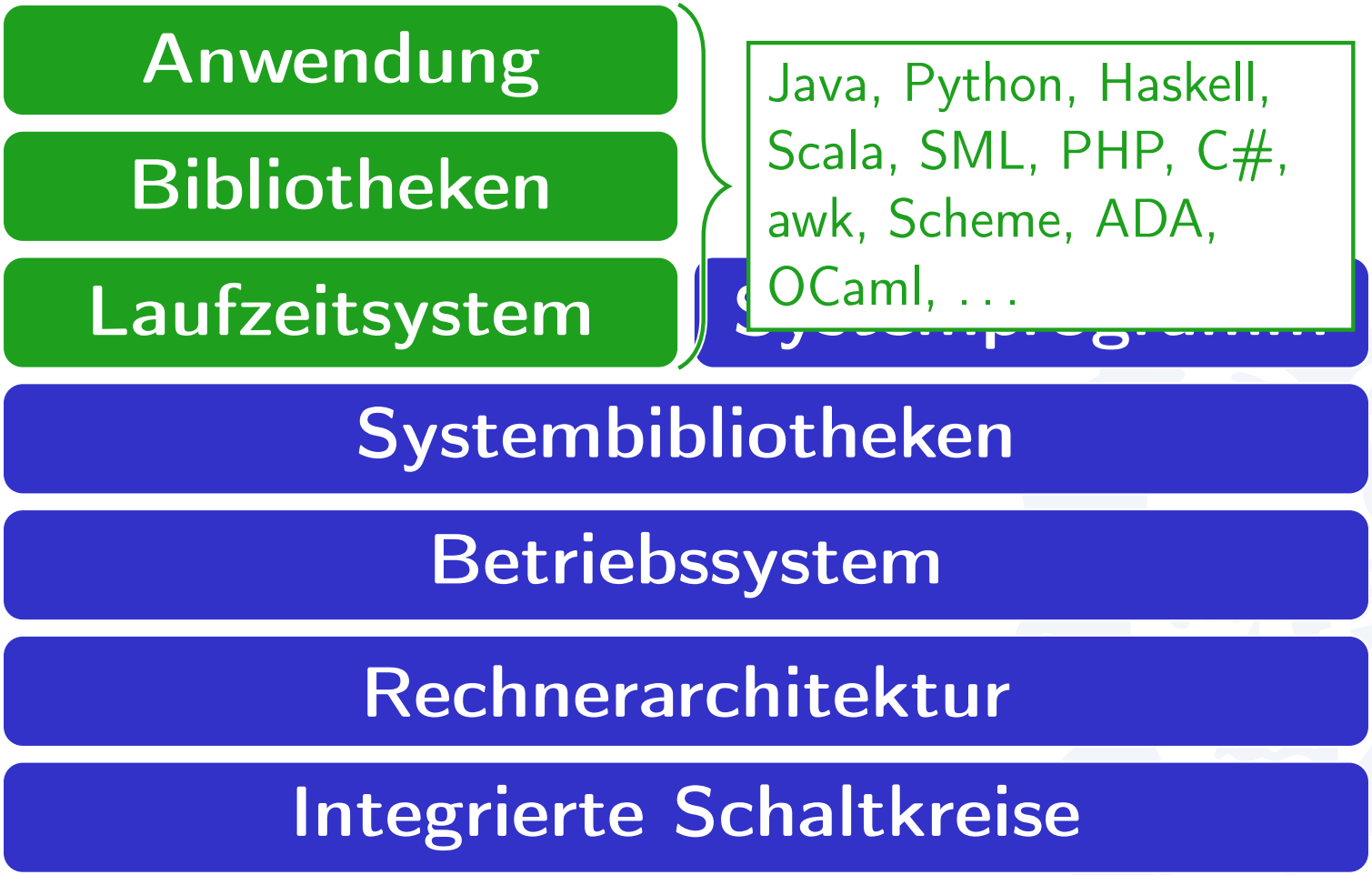
Go und seine Verwandtschaft



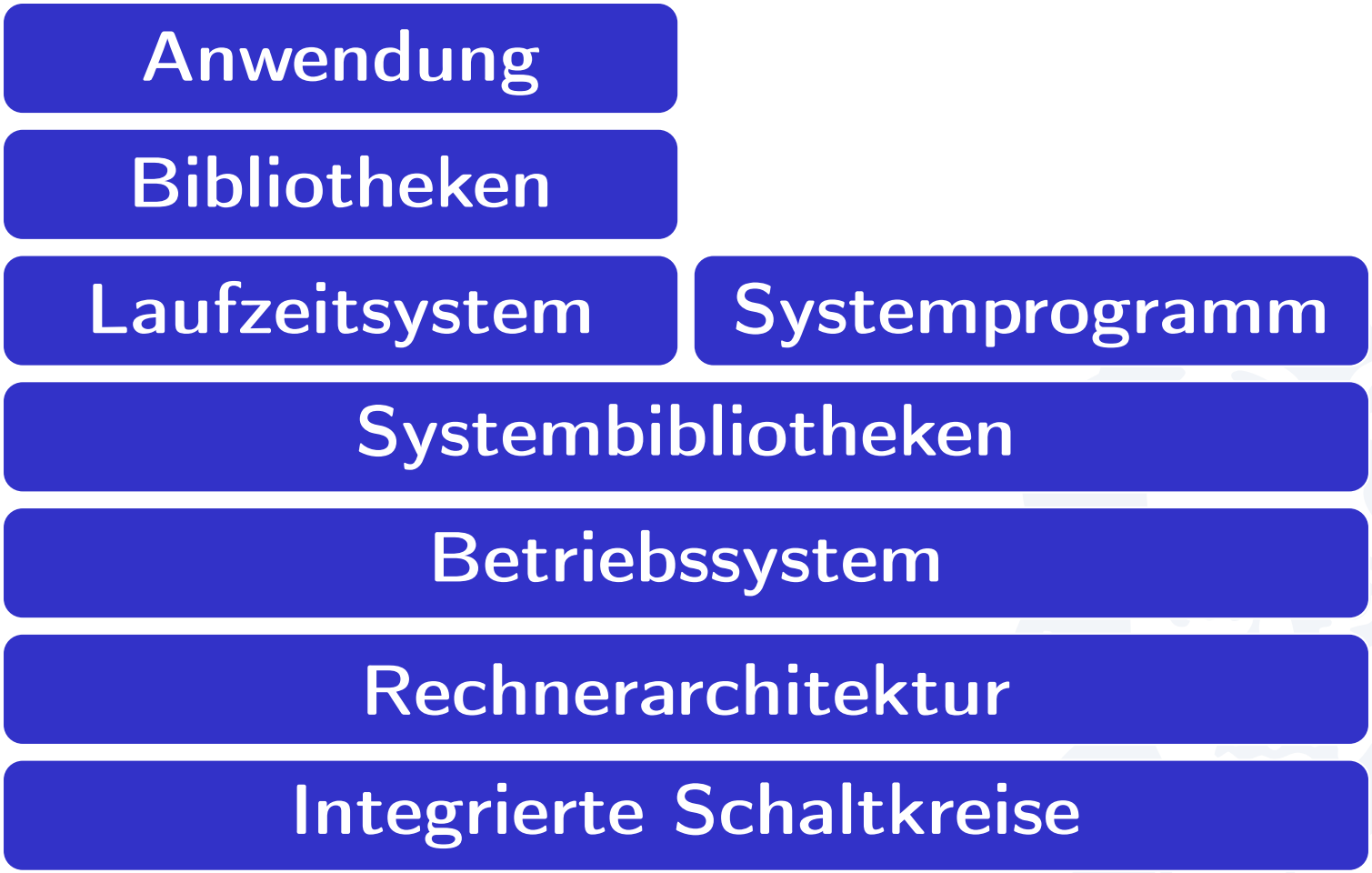
Systemprogrammierung



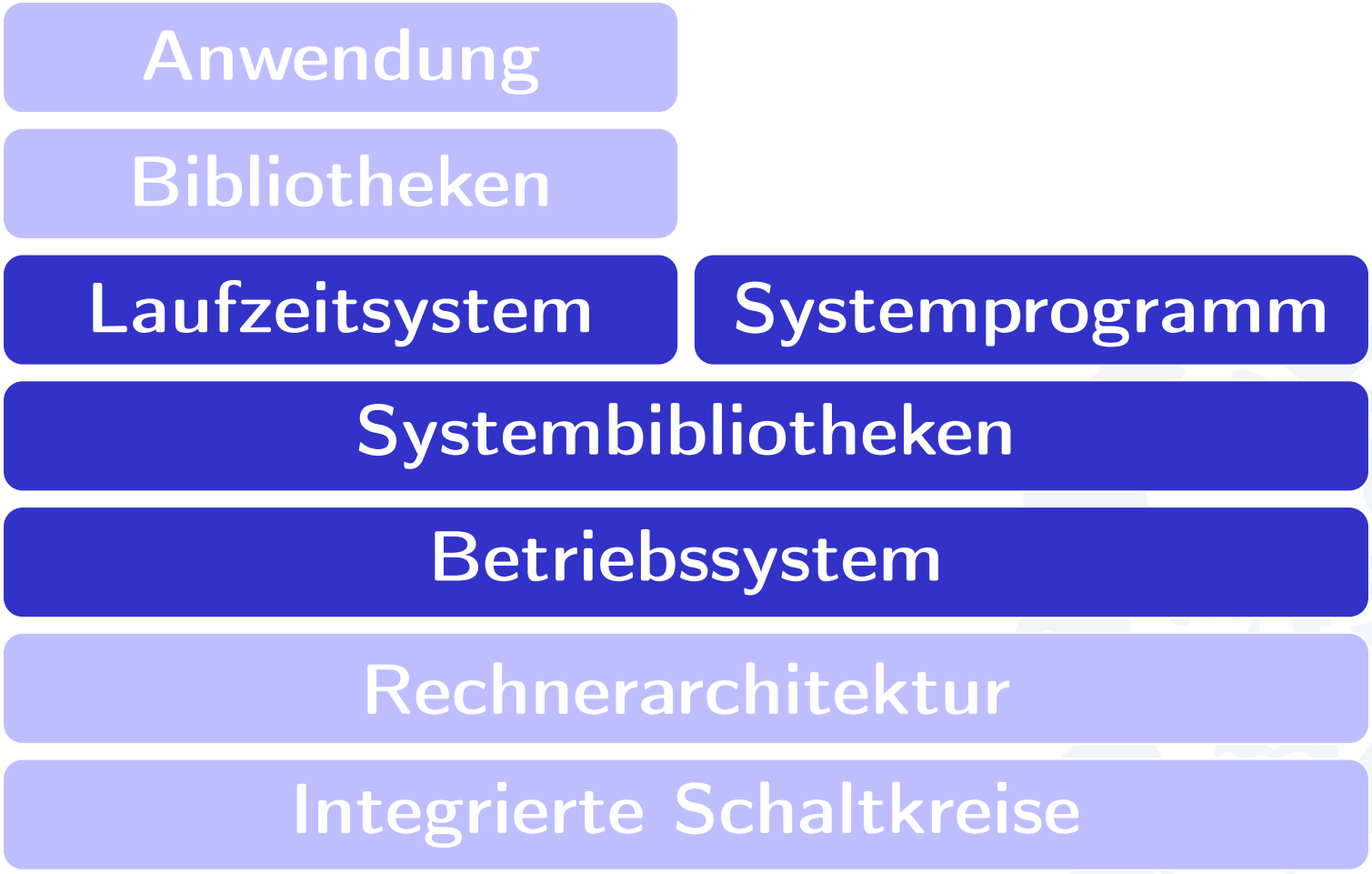
Systemprogrammierung



Systemprogrammierung



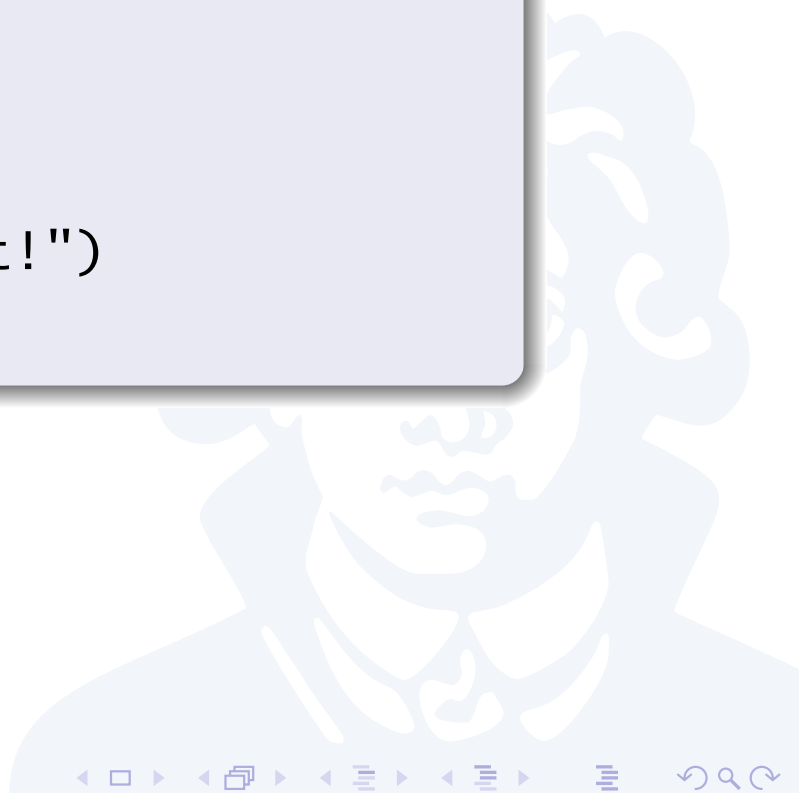
Systemprogrammierung



Hallo, Welt

Go: hallo.go

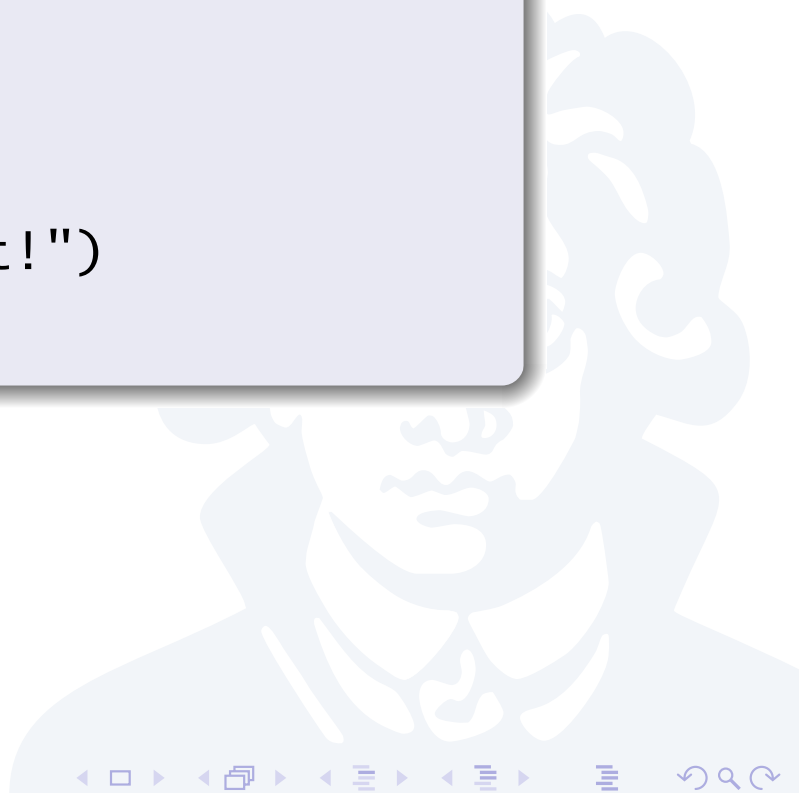
```
package main

import "fmt"

func main() {
    fmt.Println("Hallo, Welt!")
}
```



Hallo, Welt

Go: hallo.go

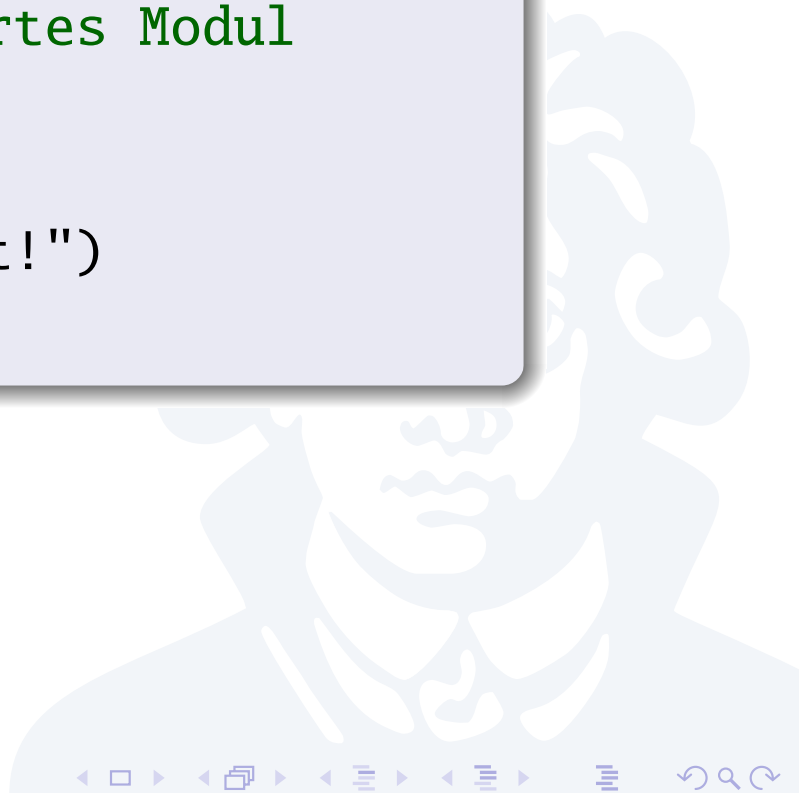
```
package main    // Paketname

import "fmt"

func main() {
    fmt.Println("Hallo, Welt!")
}
```



Hallo, Welt

Go: hallo.go

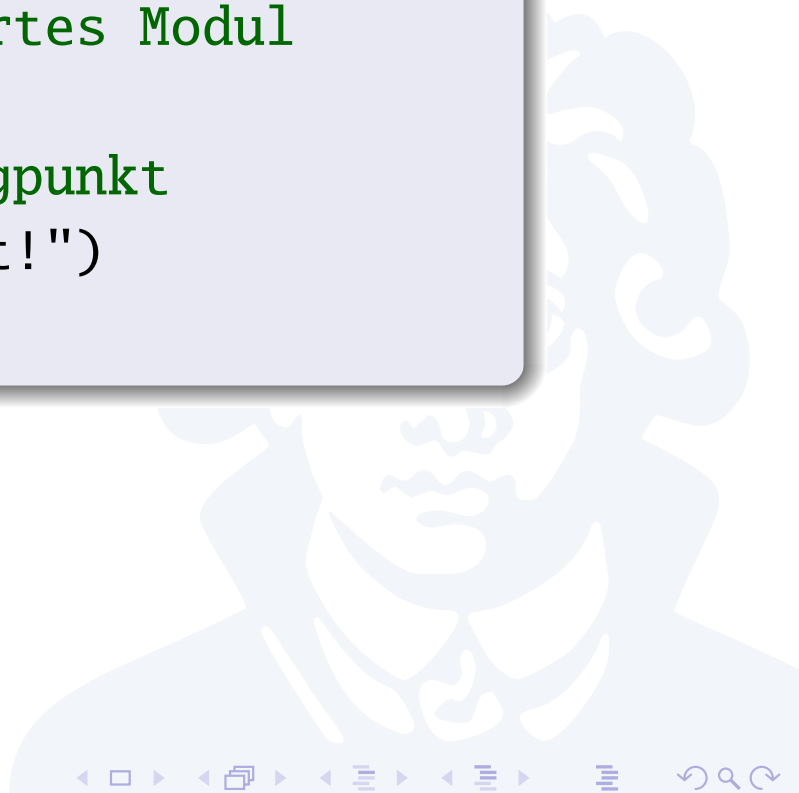
```
package main    // Paketname

import "fmt"    // Importiertes Modul

func main() {
    fmt.Println("Hallo, Welt!")
}
```



Hallo, Welt

Go: hallo.go

```
package main    // Paketname

import "fmt"    // Importiertes Modul

func main() {   // Einsprungpunkt
    fmt.Println("Hallo, Welt!")
}
```



Hallo, Welt

Go: hallo.go

```go
package main    // Paketname

import "fmt"    // Importiertes Modul

func main() {   // Einsprungpunkt
    fmt.Println("Hallo, Welt!")
}
```

```
user@host:~/projects$ go run hallo.go
Hallo, Welt!
```

Die ‘`main`’-Funktion (1/2)

FunctionDecl ::= ‘`func`’ *id* ⟨*Signature*⟩ ‘`{`’ ⟨*StatementList*⟩ ‘`}`’

- Schlüsselwort **`func`** definiert Funktion
- Funktion `main` in allen `main`-Paketen kann von außen aufgerufen werden (Einsprungpunkt)

Go: hallo.go

```
func main() {
    fmt.Println("Hallo, Welt!")
}
```

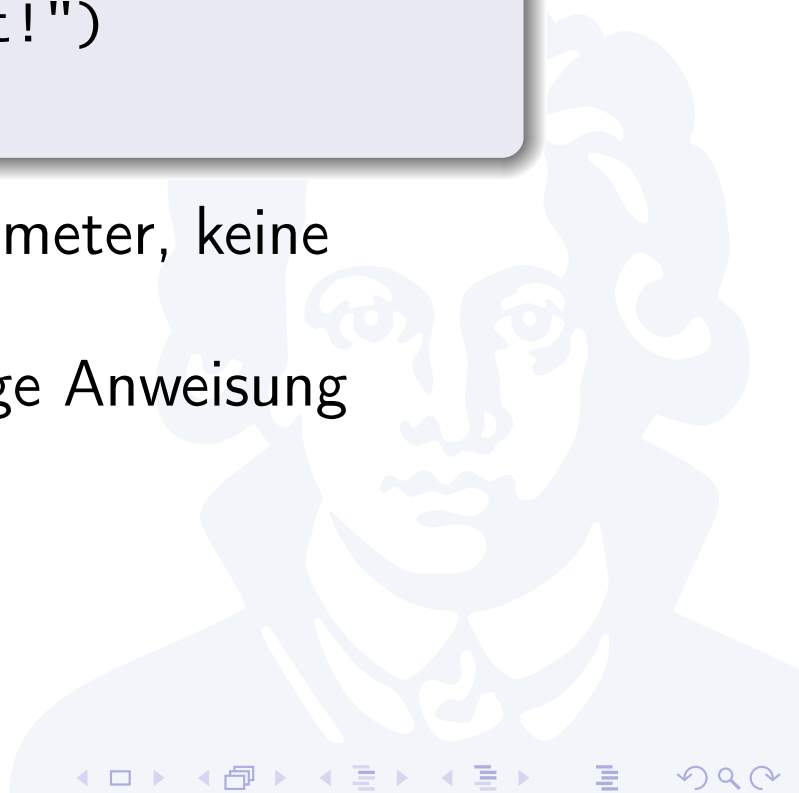
Die ‘main’-Funktion (2/2)

FunctionDecl ::= ‘func’ *id* ⟨*Signature*⟩ ‘{’ ⟨*StatementList*⟩ ‘}’

Go: hallo.go

```
func main() {
    fmt.Println("Hallo, Welt!")
}
```

- ⟨*Signature*⟩ ist hier `()` (keine Parameter, keine Rückgabewerte)
- ⟨*StatementList*⟩ ist hier die alleinige Anweisung `fmt.Println("Hallo, Welt!")`



Die ‘main’-Funktion (2/2)

FunctionDecl ::= ‘func’ *id* ⟨*Signature*⟩ ‘{’ ⟨*StatementList*⟩ ‘}’

Go: hallo.go

```
func main() {
    fmt.Println("Hallo, Welt!")
}
```

- ⟨*Signature*⟩ ist hier `()` (keine Parameter, keine Rückgabewerte)
- ⟨*StatementList*⟩ ist hier die alleinige Anweisung `fmt.Println("Hallo, Welt!")`
  - `fmt.Println` : Funktion aus dem `fmt`-Paket (Textausgabe an Konsole)

Die ‘main’-Funktion (2/2)

FunctionDecl ::= ‘func’ *id* ⟨*Signature*⟩ ‘{’ ⟨*StatementList*⟩ ‘}’

Go: hallo.go

```
func main() {
    fmt.Println("Hallo, Welt!")
}
```

- ⟨*Signature*⟩ ist hier `()` (keine Parameter, keine Rückgabewerte)
- ⟨*StatementList*⟩ ist hier die alleinige Anweisung `fmt.Println("Hallo, Welt!")`
  - `fmt.Println` : Funktion aus dem `fmt`-Paket (Textausgabe an Konsole)
  - `"Hallo, Welt!"` : Zeichenkette (`string`)

Die ‘main’-Funktion (2/2)

FunctionDecl ::= ‘func’ *id* ⟨*Signature*⟩ ‘{’ ⟨*StatementList*⟩ ‘}’

Go: hallo.go

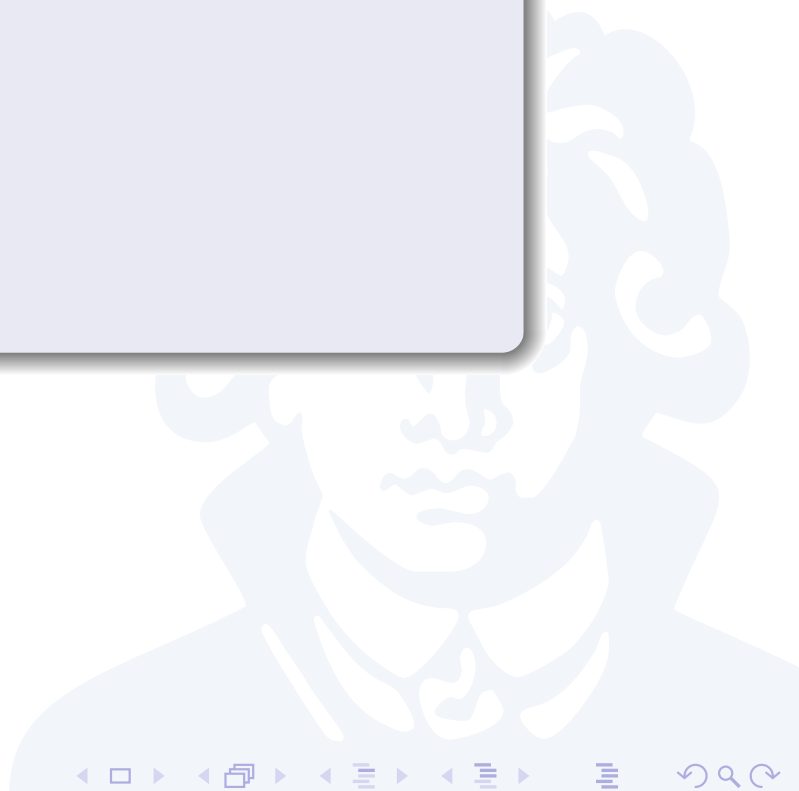
```
func main() {
    fmt.Println("Hallo, Welt!")
}
```

- ⟨*Signature*⟩ ist hier `()` (keine Parameter, keine Rückgabewerte)
- ⟨*StatementList*⟩ ist hier die alleinige Anweisung `fmt.Println("Hallo, Welt!")`
  - `fmt.Println` : Funktion aus dem `fmt`-Paket (Textausgabe an Konsole)
  - `"Hallo, Welt!"` : Zeichenkette (`string`)
  - `fmt.Println("Hallo, Welt!")` : Funktionsaufruf

Variablen

Go: i.go

```
package main

import "fmt"

func main() {
    var i int = 23
    fmt.Println("i=", i)
}
```

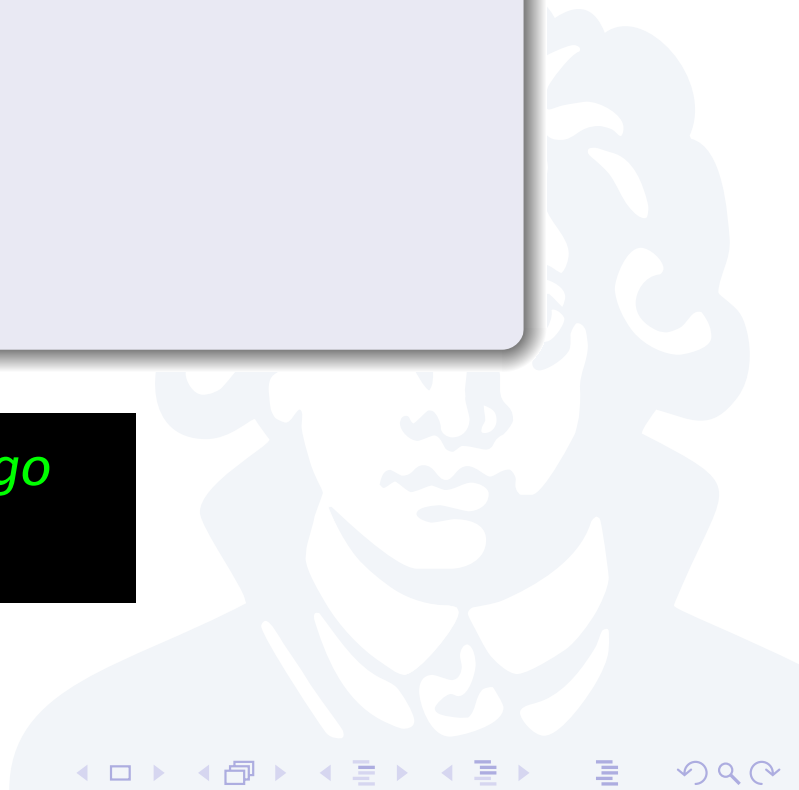


Variablen

Go: i.go

```
package main

import "fmt"

func main() {
    var i int = 23
    fmt.Println("i=", i)
}
```

```
user@host:~/projects$ go run i.go
i= 23
```



Variablen

Go: i.go

```go
package main

import "fmt"

func main() {
    var i int = 23
    fmt.Println("i=", i)
}
```

```
user@host:~/projects$ go run i.go
i= 23
```

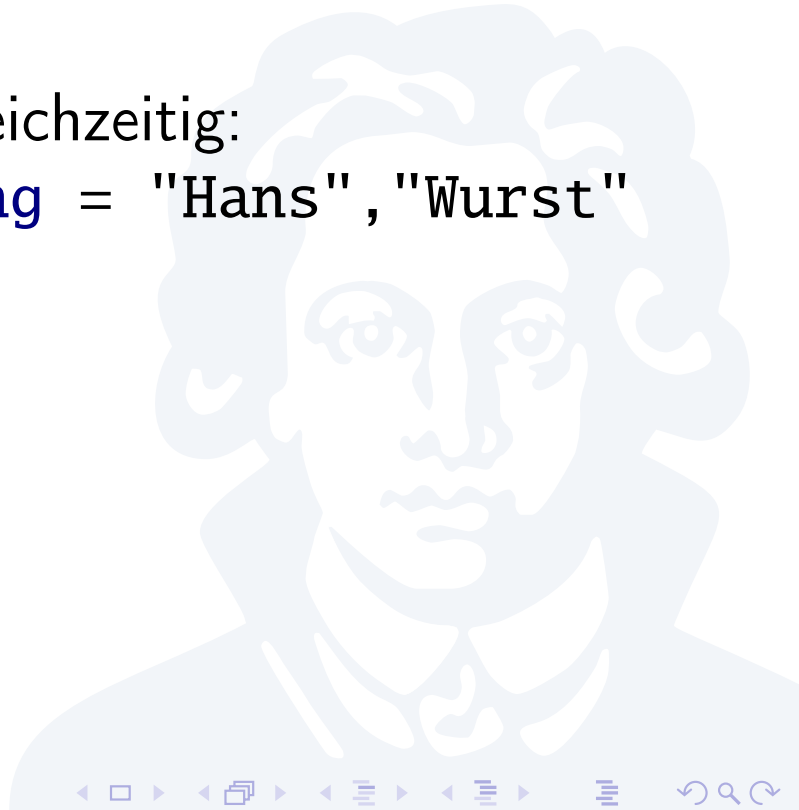
Variablen in Go sind *statisch* getypt

Deklaration von Variablen

VarDecl ::= ‘`var`’ ⟨*IdList*⟩ ⟨*Type*⟩ ‘=’ ⟨*ExpressionList*⟩

IdList ::= *id* [‘,’ *IdList*]

- Deklaration mehrerer Variablen gleichzeitig:
  ```
  var vorname, nachname string = "Hans","Wurst"
  ```

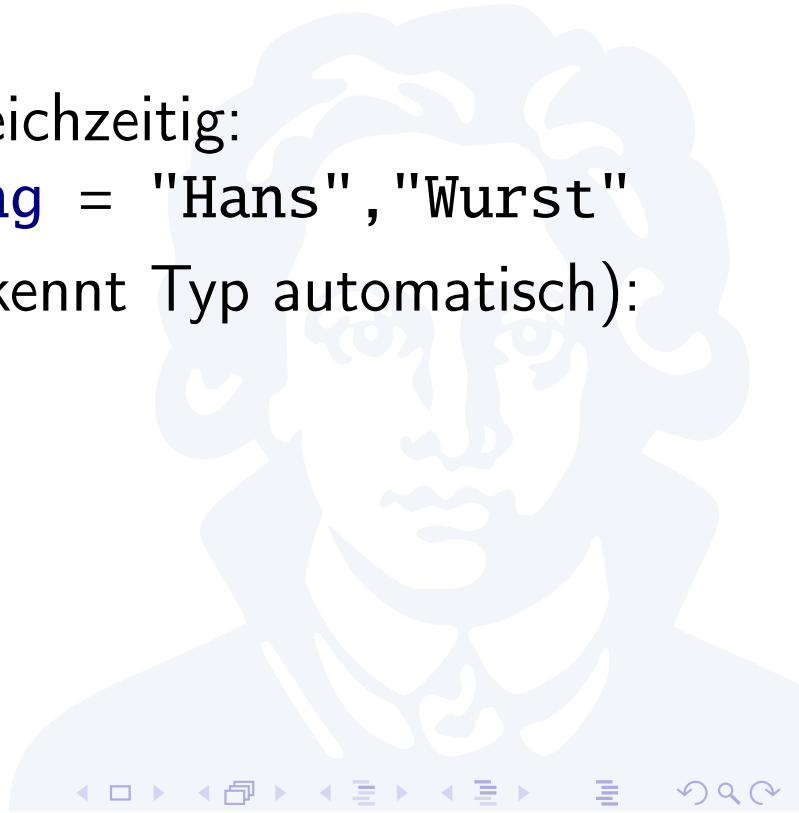


Deklaration von Variablen

VarDecl ::= ‘`var`’ ⟨*IdList*⟩ [⟨*Type*⟩] ‘=’ ⟨*ExpressionList*⟩

IdList ::= *id* [‘,’ *IdList*]

- Deklaration mehrerer Variablen gleichzeitig:
  ```
  var vorname, nachname string = "Hans","Wurst"
  ```
- Lokale Typinferenz (Übersetzer erkennt Typ automatisch):
  ```
  var pi = 3.14159265357989
  ```



Deklaration von Variablen

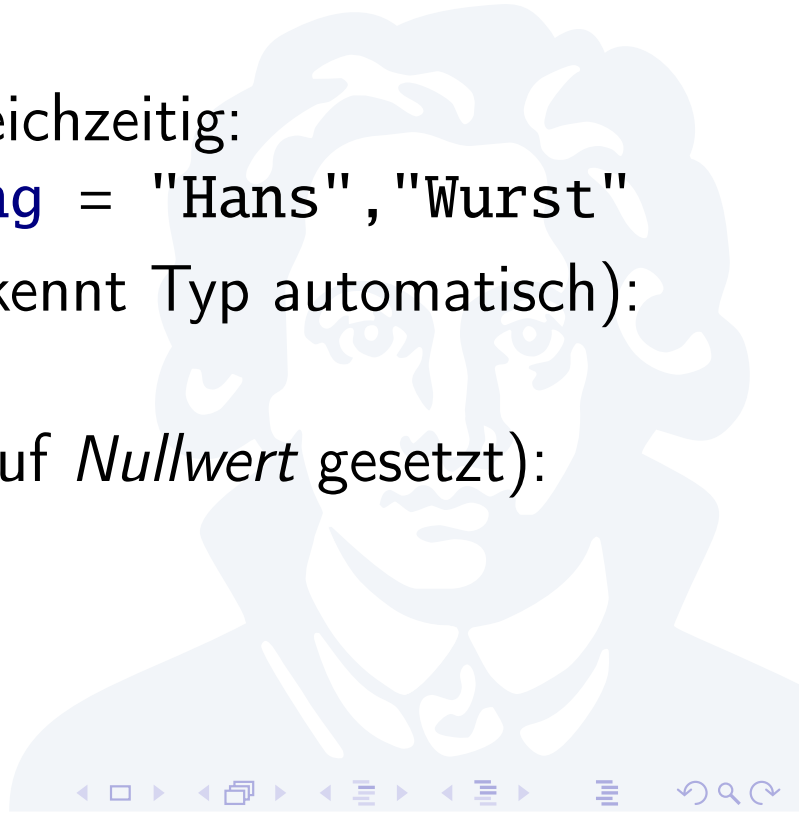
$$
\begin{array}{lcl}
\textit{VarDecl} & ::= & \text{`}\texttt{var}\text{'}\ \langle \textit{IdList} \rangle\ [\ \langle \textit{Type} \rangle\ ]\ \text{`}=\text{'}\ \langle \textit{ExpressionList} \rangle \\
& | & \text{`}\texttt{var}\text{'}\ \langle \textit{IdList} \rangle\ \langle \textit{Type} \rangle \\
\\
\textit{IdList} & ::= & \textit{id}\ [\ \text{`,'}\ \textit{IdList}\ ]
\end{array}
$$

- Deklaration mehrerer Variablen gleichzeitig:
  ```
  var vorname, nachname string = "Hans","Wurst"
  ```
- Lokale Typinferenz (Übersetzer erkennt Typ automatisch):
  ```
  var pi = 3.14159265357989
  ```
- Deklaration ohne Initialisierung (Auf *Nullwert* gesetzt):
  ```
  var x string
  ```

Deklaration von Variablen

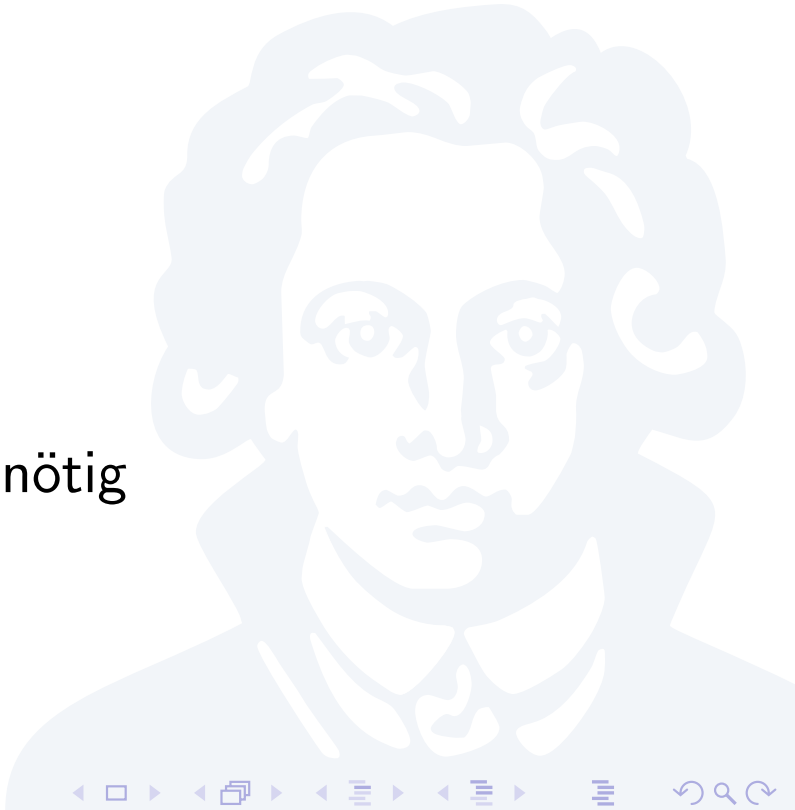
VarDecl	::=	'var' ⟨*IdList*⟩ [⟨*Type*⟩] '=' ⟨*ExpressionList*⟩
	|	'var' ⟨*IdList*⟩ ⟨*Type*⟩
ShortVarDecl	::=	⟨*IdList*⟩ ':=' ⟨*ExpressionList*⟩
IdList	::=	*id* [',' *IdList*]

- Deklaration mehrerer Variablen gleichzeitig:
  ```
  var vorname, nachname string = "Hans","Wurst"
  ```
- Lokale Typinferenz (Übersetzer erkennt Typ automatisch):
  ```
  var pi = 3.14159265357989
  ```
- Deklaration ohne Initialisierung (Auf *Nullwert* gesetzt):
  ```
  var x string
  ```
- Kurzform mit lokaler Typinferenz:
  ```
  x, y := 42, 23
  ```



Statische Typisierung

- Jede Variable hat einen *statischen Typ*
- Muß zur Übersetzungszeit bekannt sein
- Vorteile:
  - Effizientere Ausführung
  - Schnellere Übersetzung
  - Frühe Fehlererkennung
- Nachteile:
  - Frühe Fehlererkennung
  - In einigen Fällen mehr Notation nötig



Einige Basistypen

Typ	Reichweite	Beispiel	Nullwert
`int8`	$-2^7 \ldots 2^7 - 1$	`-1`, `0x2a`	`0`
`uint8`	$0 \ldots 2^8 - 1$	`1`, `0xa0`	`0`
...	...	...	...
`int64`	$-2^{63} \ldots 2^{63} - 1$	`-1234567890`	`0`
`uint64`	$0 \ldots 2^{64} - 1$	`0x102030405060cafe`	`0`
`float32`	32-Bit Fließkomma	`1e+7`, `13.7`	`0.0`
`float64`	64-Bit Fließkomma	`1.2e+200`, `13.7`	`0.0`
`string`	Zeichenketten	`"Unĩcøδe"`, `‘xy‘`	`""`
`bool`	Wahrheitswerte	`true`, `false`	`false`

- Fließkommazahlen nach IEEE-754
- `byte` = `uint8`
- `int` entspricht entweder `int32` oder `int64`
- `uint` analog

Einige Basistypen

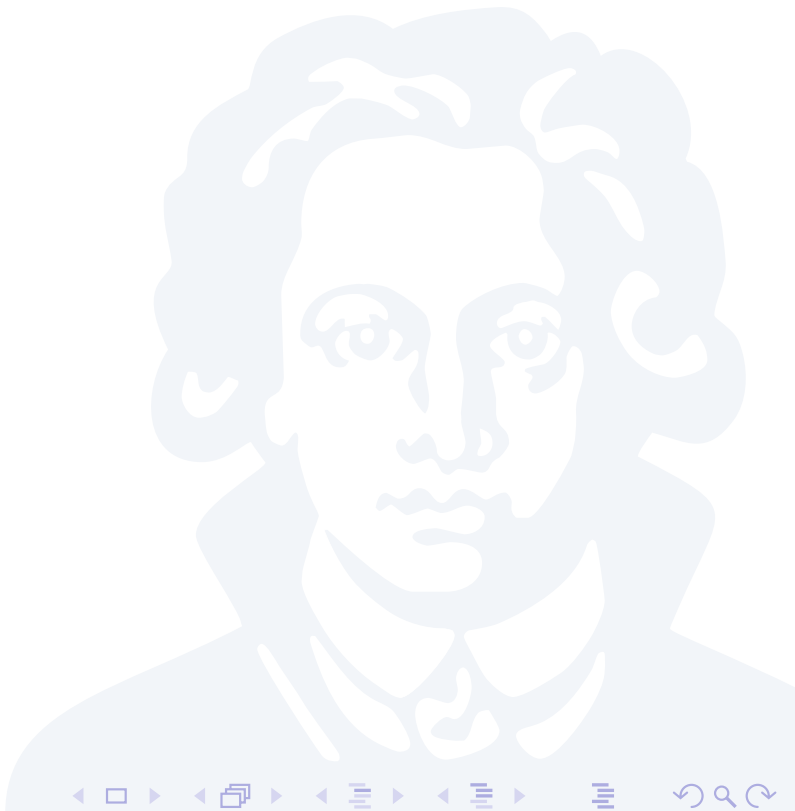
	Typ	Reichweite	Beispiel	Nullwert
Numerische Typen	`int8`	$-2^7 \ldots 2^7 - 1$	`-1`, `0x2a`	`0`
	`uint8`	$0 \ldots 2^8 - 1$	`1`, `0xa0`	`0`
	...	...	...	...
	`int64`	$-2^{63} \ldots 2^{63} - 1$	`-1234567890`	`0`
	`uint64`	$0 \ldots 2^{64} - 1$	`0x102030405060cafe`	`0`
	`float32`	32-Bit Fließkomma	`1e+7`, `13.7`	`0.0`
	`float64`	64-Bit Fließkomma	`1.2e+200`, `13.7`	`0.0`
	`string`	Zeichenketten	`"Unĩcøδe"`, `‘xy‘`	`""`
	`bool`	Wahrheitswerte	`true`, `false`	`false`

- Fließkommazahlen nach IEEE-754
- `byte` = `uint8`
- `int` entspricht entweder `int32` oder `int64`
- `uint` analog

Arithmetische Operatoren

Arithmetische Operatoren für numerische Typen

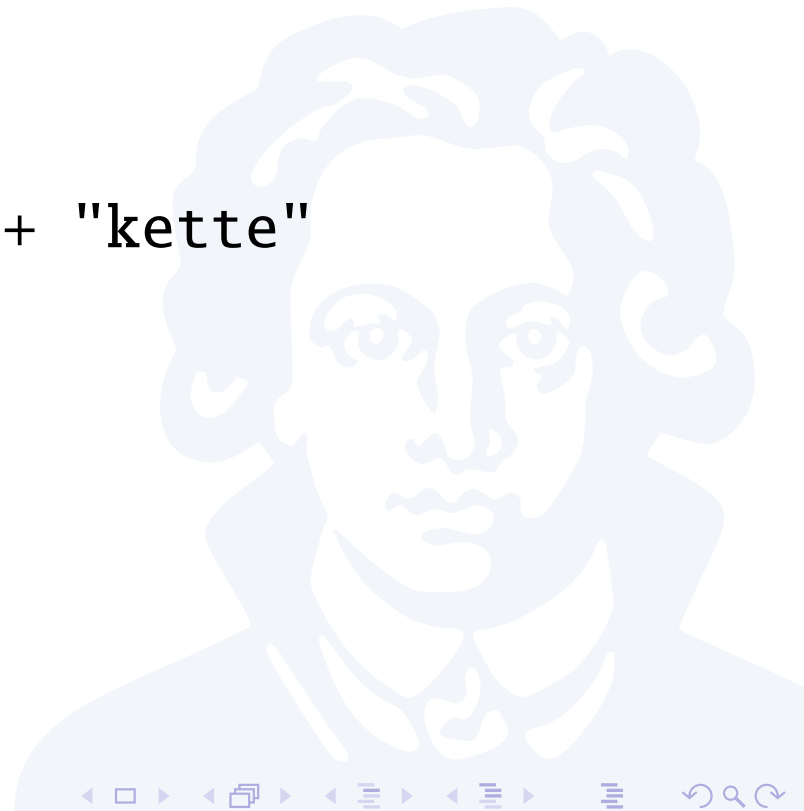
- -, *, /, +
- %: Modulo (Divisionsrest)



Arithmetische Operatoren

Arithmetische Operatoren für numerische Typen

- -, *, /, +
- %: Modulo (Divisionsrest)
- +: Auch für `string`: `"Zeichen" + "kette"`



Vergleiche

- Ergebnis ist `bool`: `true` (wahr) oder `false` (falsch)
- Vergleiche über beliebige Typen:
  - Gleichheit:

    ```
    a == b
    ```

  - Ungleichheit:

    ```
    a != b
    ```

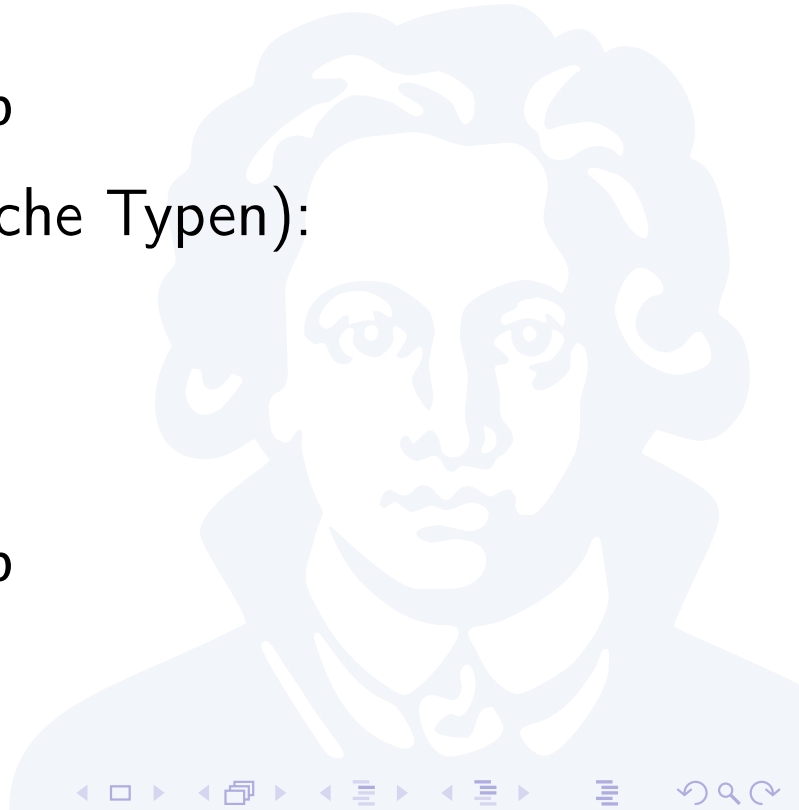
- Arithmetische Vergleiche (numerische Typen):
  - Kleiner-als:

    ```
    a < b
    ```

  - Kleiner-oder-gleich:

    ```
    a <= b
    ```

  - Analog >, >=



Beisipiel mit **for**-Schleife

Go: schleife.go

```
package main

import "fmt"

func main() {
    for i := 0; i <= 3; i = i + 1 {
        fmt.Println(i)
    }
}
```

Beisipiel mit **for**-Schleife

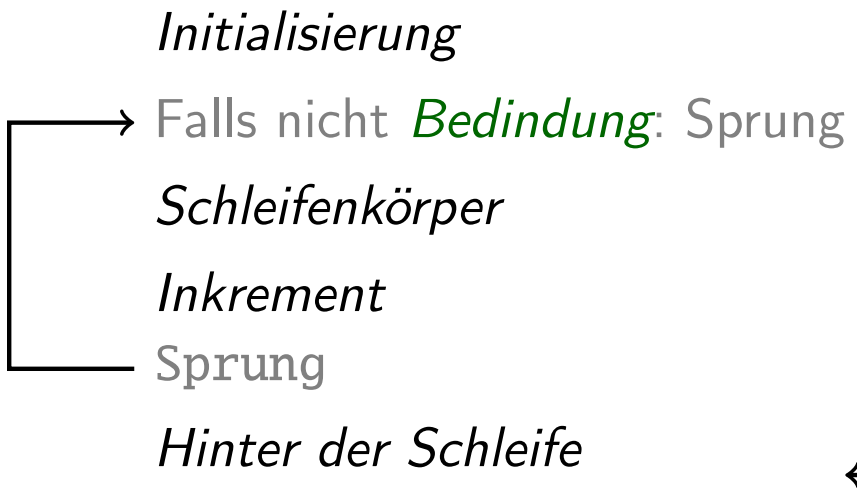
Go: schleife.go

```
package main

import "fmt"

func main() {
    for i := 0; i <= 3; i = i + 1 {
        fmt.Println(i)
    }
}
```

```
user@host:~/projects$ go run schleife.go
0
1
2
3
```

Einfache `for`-Schleifen

```
for Initialisierung; Bedingung; Inkrement {
    Schleifenkörper
}
Hinter der Schleife
```



Alternativen zur Addition

Go: schleife.go

```
package main

import "fmt"

func main() {
    for i := 0; i <= 3; i = i + 1 {
        fmt.Println(i)
    }
}
```

```
user@host:~/projects$ go run schleife.go
0
1
2
3
```

Alternativen zur Addition

Go: schleife.go

```
package main

import "fmt"

func main() {
    for i := 0; i <= 3; i += 1 {
        fmt.Println(i)
    }
}
```

```
user@host:~/projects$ go run schleife.go
0
1
2
3
```

Alternativen zur Addition

Go: schleife.go

```
package main

import "fmt"

func main() {
    for i := 0; i <= 3; i++ {
        fmt.Println(i)
    }
}
```

```
user@host:~/projects$ go run schleife.go
0
1
2
3
```

Arithmetik und Zuweisungen

Go: Arithmetik

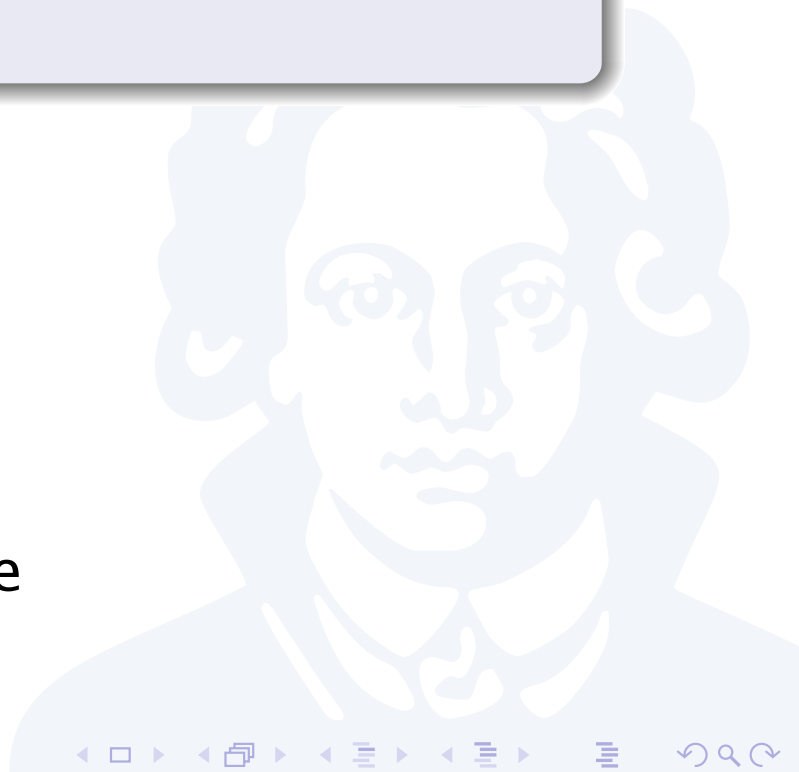
```
var i int
i = 13  // Zuweisung: 'i' wird überschrieben
i = i * 2 - 6 / 3 * 2
k := 10
k += i
```

- Sei $\otimes$ ein Operator:

  $$\texttt{v} \otimes= \texttt{e}$$

  ist (fast) äquivalent zu

  $$\texttt{v} = \texttt{v} \otimes \texttt{e}$$



Arithmetik und Zuweisungen

Go: Arithmetik

```
var i int
i = 13  // Zuweisung: 'i' wird überschrieben
i = i * 2 - 6 / 3 * 2
k := 10
k += i
```

- Sei $\otimes$ ein Operator:

  $$\texttt{v} \otimes= \texttt{e}$$

  ist (fast) äquivalent zu

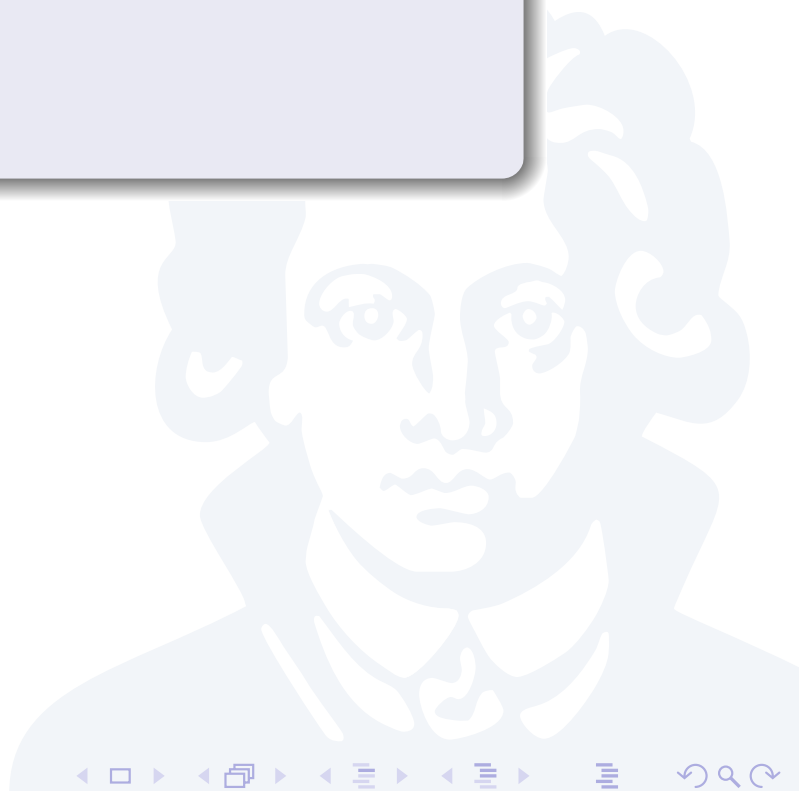
  $$\texttt{v} = \texttt{v} \otimes \texttt{e}$$

- `v++` entspricht `v += 1`
- `v--` entspricht `v -= 1`

Bedingte Ausführung mit `if`

Go: Bedingungen

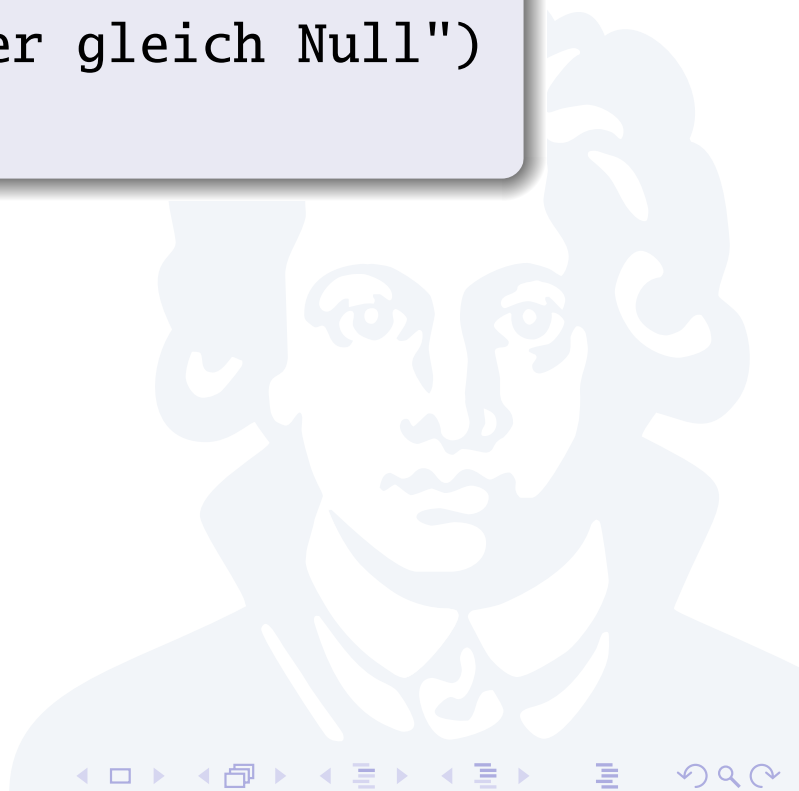
```
if i > 0 {
    fmt.Println("größer als Null")
}
```



Bedingte Ausführung mit `if`

Go: Bedingungen

```
if i > 0 {
    fmt.Println("größer als Null")
} else {
    fmt.Println("kleiner oder gleich Null")
}
```



Bedingte Ausführung mit `if`

Go: Bedingungen

```
if i > 0 {
    fmt.Println("größer als Null")
} else {
    fmt.Println("kleiner oder gleich Null")
}
```

Go: `if` mit Initialisierung

```
if x := a*b; x < 0 {
    fmt.Println("Ungleiche Vorzeichen: ", x)
}
```

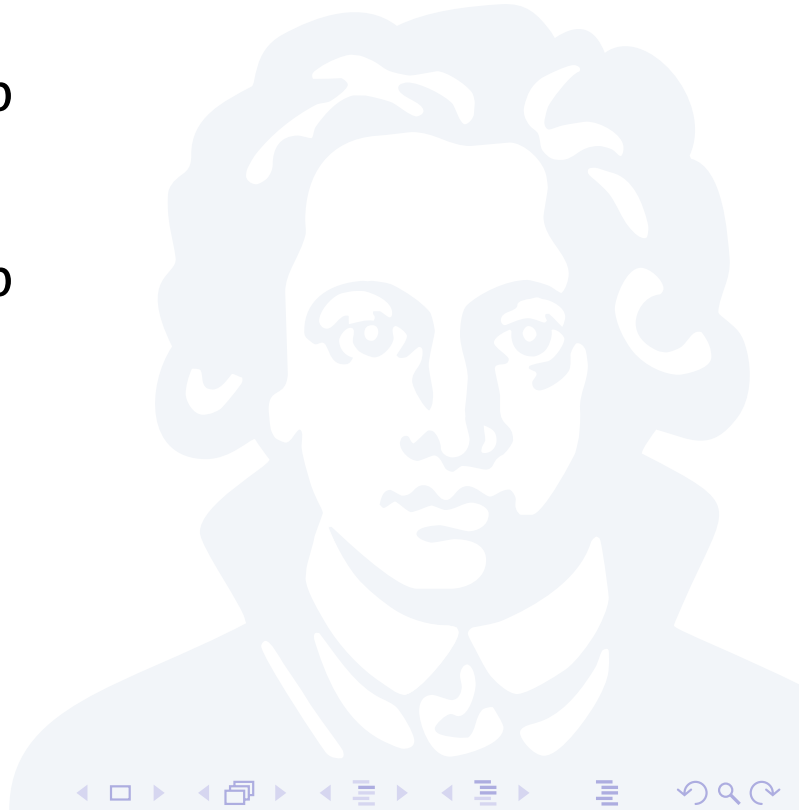
Logische Operatoren

- Logische Konnektive
  - Logisches UND:

    ```
    a && b
    ```

  - Logisches ODER:

    ```
    a || b
    ```



Zusammenfassung: Operatoren

Binäre Operatoren

5	*	/	%	<<	>>	&	&ˆ
4	+	-	|	ˆ			
3	==	!=	<	<=	>	>=	
2	&&						
1	||						

- Höhere Präzedenz bindet stärker, sonst links-assoziativ:

```
i * 2 - 6 / 3 * 2  =  (i * 2) - ((6 / 3) * 2)
```

Unäre Operatoren

- $+x$: $0 + x$
- $-x$: $0 - x$
- ˆx: Bitweise Komplement
- !x: Logisches Komplement
- &x
- <-x

Logisch Gesteuerte Schleifen

Go: GCD

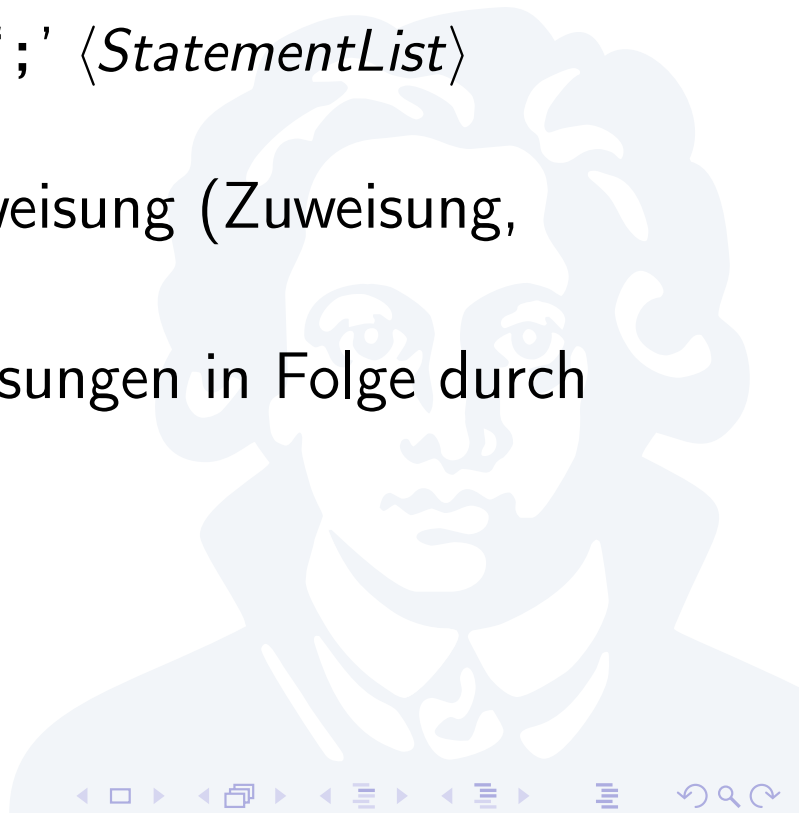
```
for a != b {
    if a > b {
        a -= b
    } else {
        b -= a
    }
}
fmt.Println(a);
```

- Alternativform von **for**: Wiederhole, solange Bedingung wahr ist
- Bedingung muß `bool`-Wert (Wahrheitswert) berechnen
- Entspricht *while*-Schleife in anderen Sprachen

Blöcke

$$
\begin{array}{lcl}
\textit{Block} & ::= & \text{`\{'} \ \langle \textit{StatementList} \rangle \ \text{`\}'} \\
\textit{StatementList} & ::= & \varepsilon \\
& | & \langle \textit{Statement} \rangle \ \text{`;'} \ \langle \textit{StatementList} \rangle
\end{array}
$$

- $\langle \textit{Statement} \rangle$ ist eine beliebige Anweisung (Zuweisung, Schleife, Block, ...)
- Ein Block führt alle inneren Anweisungen in Folge durch



Zusammenfassung: Kontrollstrukturen

$$
\begin{array}{lcl}
\textit{Block} & ::= & \text{`}\texttt{\{}\text{'}\ \langle \textit{StatementList} \rangle\ \text{`}\texttt{\}}\text{'} \\
\textit{For} & ::= & \text{`}\texttt{for}\text{'}\ [\ \langle \textit{SimpleStmt} \rangle\ ]\ \text{`}\texttt{;}\text{'}\ \langle \textit{Expression} \rangle\ \text{`}\texttt{;}\text{'}\ [\ \langle \textit{SimpleStmt} \rangle\ ]\ \langle \textit{Block} \rangle \\
& | & \text{`}\texttt{for}\text{'}\ \langle \textit{Expression} \rangle\ \langle \textit{Block} \rangle \\
& | & \text{`}\texttt{for}\text{'}\ \langle \textit{Block} \rangle \\
\textit{IfStmt} & ::= & \text{`}\texttt{if}\text{'}\ [\ \langle \textit{SimpleStmt} \rangle\ \text{`}\texttt{;}\text{'}\ ]\ \langle \textit{Expression} \rangle\ \langle \textit{Block} \rangle\ [\ \text{`}\texttt{else}\text{'}\ \langle \textit{Block} \rangle\ ] \\
& | & \text{`}\texttt{if}\text{'}\ [\ \langle \textit{SimpleStmt} \rangle\ \text{`}\texttt{;}\text{'}\ ]\ \langle \textit{Expression} \rangle\ \langle \textit{Block} \rangle\ [\ \text{`}\texttt{else}\text{'}\ \langle \textit{IfStmt} \rangle\ ]
\end{array}
$$

- Block: Folge von Anweisungen
- **`if`**: Bedingte Ausführung
  - **`else`**: Alternative (entweder Block oder Folge-**`if`**)
- **`for`**: Wiederholung
  - **`for`** `{` *A* `}`: Wiederhole *A*
  - **`for`** *e* `{` *A* `}`: Wiederhole *A* bis *e* = `false`
  - **`for`** *Init* `;` *Test* `;` *Post* `{` *A* `}`
- **`for`** und **`if`** sind hier leicht vereinfacht

Funktionen: Beispiel

Go: funktion.go

```
package main

import "fmt"

func f(a int, b int) int {
    return a + b
}

func main() {
    fmt.Println(f(1, 2), f(2, 3))
}
```

Funktionen: Beispiel

Go: funktion.go

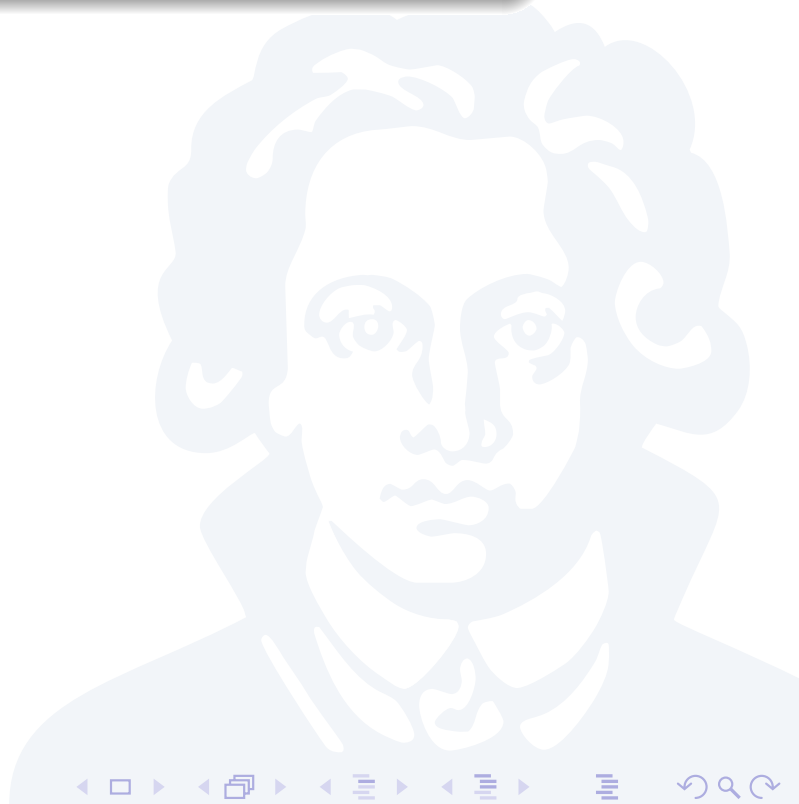
```
package main

import "fmt"

func f(a int, b int) int {
    return a + b
}

func main() {
    fmt.Println(f(1, 2), f(2, 3))
}
```

```
user@host:~/projects$ go run funktion.go
3 5
```

Funktionen mit mehreren Rückgabewerten

Go: swap1

```
func swap1(x, y int) (int, int) {
    return y, x
}
```



Funktionen mit mehreren Rückgabewerten

Go: swap1

```
func swap1(x, y int) (int, int) {
    return y, x
}
```

Äquivalent:

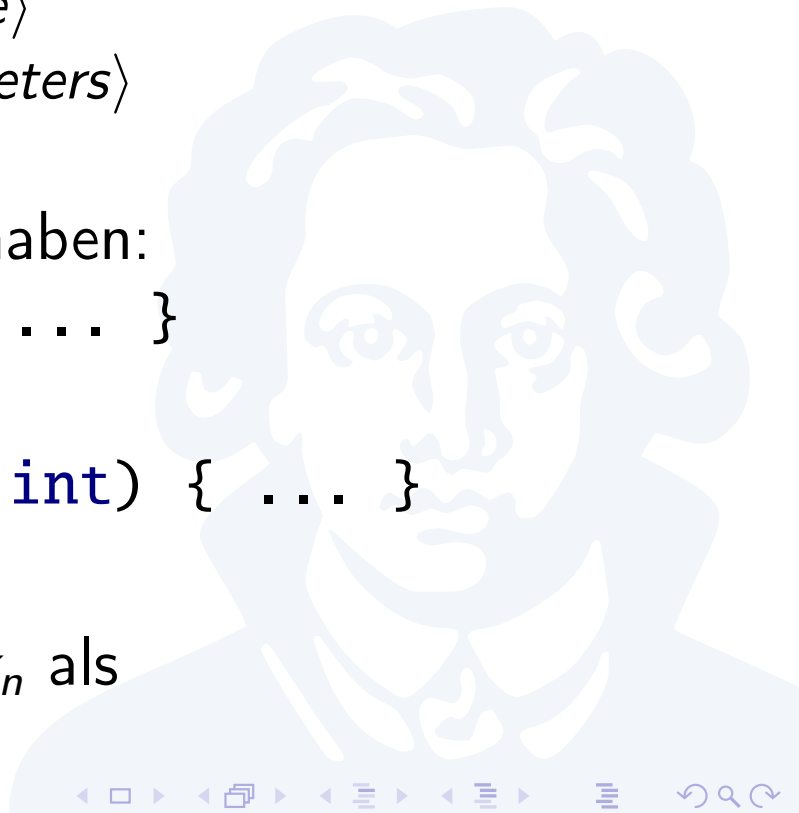
Go: swap2

```
func swap2(x, y int) (rx, ry int) {
    rx = y
    ry = x
    return
}
```

Funktionen

FunctionDecl	::=	‘**func**’ *id* ⟨*Signature*⟩ ⟨*Block*⟩
Signature	::=	⟨*Parameters*⟩ [⟨*Return*⟩]
Parameters	::=	‘(’ [⟨*ParameterList*⟩] ‘)’
ParameterList	::=	⟨*ParameterDecl*⟩ [‘,’ ⟨*ParameterList*⟩]
ParameterDecl	::=	[⟨*IdList*⟩] ⟨*Type*⟩
Return	::=	⟨*Type*⟩ | ⟨*Parameters*⟩

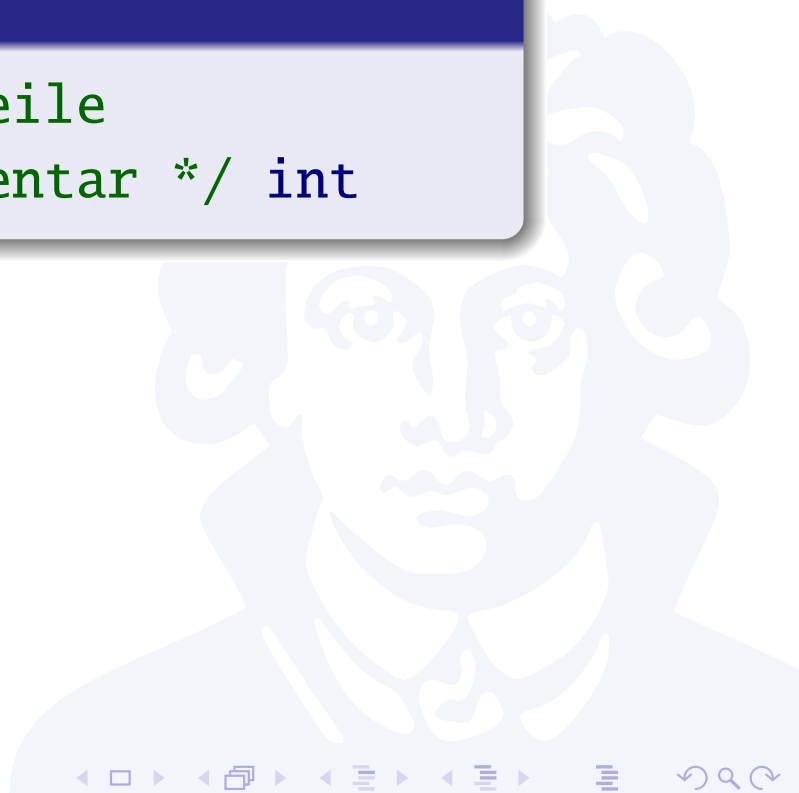
- Funktionen können Rückgabetyp haben:
  func `f (x, y int) string { ... }`
- Mehrere Rückgabewerte möglich:
  func `swap (x, y int) (int, int) { ... }`
- **return** beendet Funktion
- **return** $x_1, \ldots, x_n$ liefert $x_1, \ldots, x_n$ als Funktionsergebnisse



Kommentare

Go: kommentar.go

```
// Kommentar bis Ende der Zeile
var x /* eingebetteter Kommentar */ int
```



Semikolon

Go: semikolon.go

```
f(); g(); h()
```

ist gleichwertig zu:

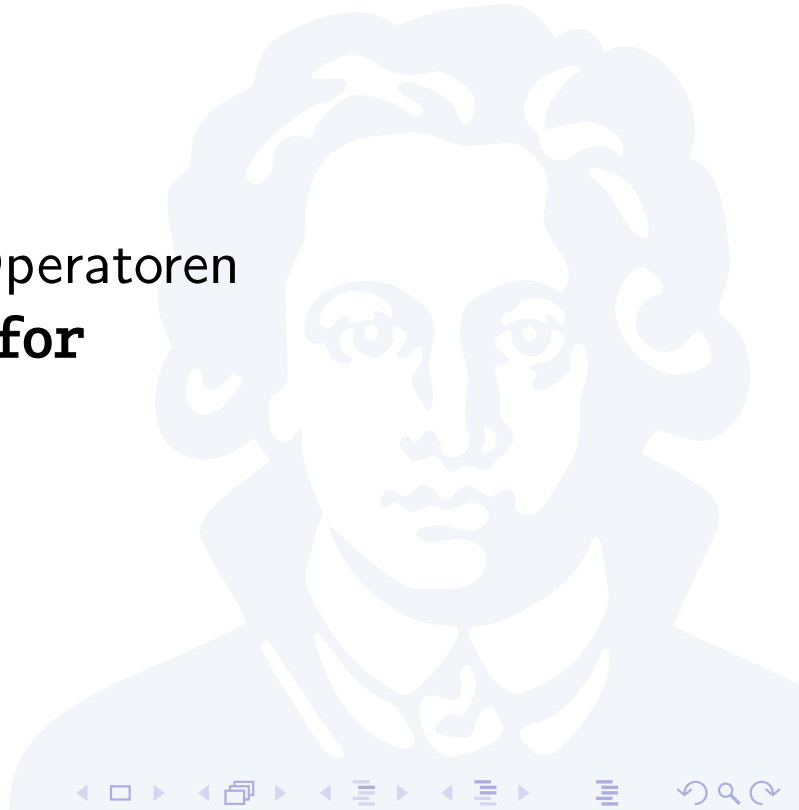
Go: kein-semikolon.go

```
f()
g()
h()
```

Die Sprache führt bei Zeilenumbrüchen *an geeigneten Stellen* automatisch ein Semikolon ein

Zusammenfassung: Grundlagen von *Go*

- Sprache für systemnahe Programmierung
- Automatische Speicherverwaltung
- Wir haben betrachtet:
  - Variablen
  - Arithmetik
  - Vergleichsoperatoren, logische Operatoren
  - Kontrollstrukturen: Blöcke, **`if`**, **`for`**
  - Funktionen: **`func`**, **`return`**



Ausblick

1. Einführung
2. Datenstrukturen und Module
3. Typsystem
4. Nebenläufige Ausführung
5. Bibliotheken und Fehlerbehandlung