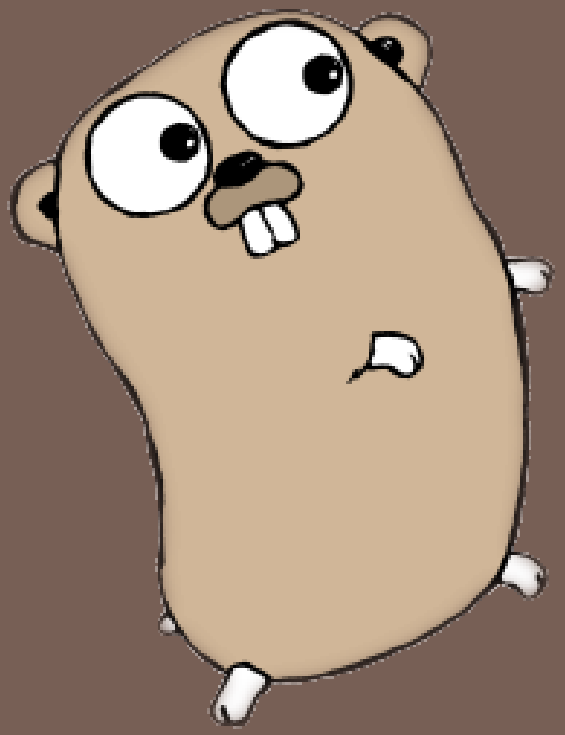

THE GO PROGRAMMING LANGUAGE

Part 1:

Michael Karnutsch & Marko Sulejic

Gliederung

- Geschichte / Motivation
- Compiler
- Formatierung, Semikolons
- Variablen, eigene Typen
- Kontrollstrukturen
- Funktionen, Methoden
- Packages
- Pointer vs. Values
- Interfaces, Embedding
- Quelle(n)

Geschichte

- Robert Griesemer, Rob Pike, Ken Thompson
- Januar 2008 – Compilerbau
- Ian Lance Taylor – GCC front end
- Russ Cox – Ende 2008
- Gründe:
  - Effiziente Kompilierung
  - Effiziente Ausführung
  - Einfache Programmierung
  - GC, Nebenläufigkeit, …

Compiler

- 6g/6l
  - Experimentell
  - Code OK
  - schnell übersetzt
  - Nicht gcc - linkbar
- Gccgo
  - Mehr traditionell
  - Code meist besser
  - Nicht so schnell
  - Gcc – linkbar
- Beide existieren für AMD64, 368, ARM

Formatierung, Semikolons

- gofmt
  - automatisches anpassen der Abstände (Tab, Leerzeichen)
- Keine Semikolons im Code
- Ausnahme: For – Schleifen
- Kein „newline“ vor Klammern
- Beispiel

Variablen

- Schlüsselwort „var“
  - `var a uint64 = 0`
  - `a := uint64(0)`
  - `i := 0x1234 // Standardtyp: int`
  - `var j int = 1e6 // int (10 hoch 6)`
  - `x := 1.5 // float`
  - `b := 3/2 // int`
  - `b := 3./2. // float`
  - `a, b, c, d := 123, 2.12, PI, „myst“`

Variablen, Typen

```
var (
    a int
    b float
    c  string = „Text“
)
var p struct {x, y float}
```

- Schlüsselwort „type“
  - `type Point struct {x, y float}`
  - `var p Point`
  - `p = Point { 7.2, 8.4 }`

Kontrollstrukturen - IF

```
if x > 0 {
    return y
}
```

```
if x := os.Open(foo.txt); x != nil {
    return true;
}
```

Kontrollstrukturen - FOR

- **for init; condition; post { }**
- ```
  sum := 0
   for i := 0; i < 10; i++ {
        sum += i
   }
  ```
- **for condition { }**
- ```
  for i<10 {
   i++
   }
  ```
- ```
  for { }
  ```
  ```

Kontrollstrukturen - FOR

- Iterieren über Arrays, Slices, Channels, Strings, Maps
  - Range

```
for pos, char := range „ABC“ {     //key, value
    fmt.Printf(„char %c starts at pos %d“, char, pos)
}
```

Kontrollstrukturen

- Keine „while“ oder „do – while“
- „switch“ wie gewohnt
- Cases können zusammengefasst werden
  - `Case ‘?‘, ‘a‘, ‘d‘, ‘p‘, ‘r‘, ‘ö‘ :`

Funktionen

- Schlüsselwort „func“
  - `func square (f float) float { return f*f }`
- Multiple returns:
  - ```
    func MySqrt (f float) (float, bool) {
      if f>=0 { return math.Sqrt(f), true}
       return 0, false
    }
    ```
    ```

Funktionen – Named Returns

```
func MySqrt(f float) (v float, ok bool) {
   if f>=0 { v,ok = math.Sqrt(f), true }
   else { v,ok = 0,false }
   return v,ok
}

func MySqrt(f float) (v float, ok bool) {
   if f>= 0 { v,ok = math.Sqrt(f), true }
   return v,ok
}
```

Funktionen – Empty Returns

```
func MySqrt(f float) (v float, ok bool) {
    if f >= 0 { v,ok = math.Sqrt(f), true }
    return
}
```

```
func MySqrt(f float) (v float, ok bool) {
    if f < 0 { return }
    return math.Sqrt(f), true
}
```

Methoden auf Strukturen

```
type Point struct {x , y float}
```

```
func (p *Point) Abs() float {
    return math.Sqrt (p.x*p.x + p.y*p.y)
 }
```

- Beispiel:
  - `p := Point {3 , 4}`
  - `i := p.Abs()`

Methoden auf Nicht-Strukturen

```
type IntVector []int

func (v IntVector) Sum() (s int) {
    for i, x := range v {
    s += x
    }
    return
}

IntVector{1,2,3}.Sum()
```

Defer

- Schlüsselwort „defer“ deklariert Funktionen, die nach einem „return“ ausgeführt werden.
- Nützlich um files, channels, mutexes, db-connections, usw. zu schließen oder Fehlerfälle zu behandeln.

```
func (file File) myst (byte, … ) {
    defer file.close()
    file.write() …
    return
}
```

Packages

- Go besteht aus einer Menge von Packages
- Jedes Programm ist selbst ein Package
  - Verwendet bestehende Packages
  - Kann aus mehreren Source-files bestehen
- Zugriffe auf importierte Packages erfolgen über „qualified identifier“ : `packagename.Itemname`

Packages - Aufbau

- Jedes Sourcefile enthält:
  - Eine Packetdeklaration:
    - `package fmt`
  - Importdeklarationen:
    - `import “fmt“`
  - Optionale Variablen, Funktionen, … Deklarationen
    - global
    - auf Packetebene

Packages - Sichtbarkeit

- Unterschied zu C/Java/...
  - kein `extern`, `static`, `public`, `private`, `default`, ...
- Innerhalb eines Packages sind alle Variablen, Funktionen, Types, etc. sichtbar.
- Importers von Packages können nur auf Variablen, Funktionen, etc. zugreifen, falls diese mit einem Großbuchstaben deklariert sind.

Packages – Main, Initialisierung

- Jedes Go Programm beinhaltet ein Package „main“
  - enthält den Einsprungspunkt (main-Methode)
  - main-Methode hat keine Parameter und keinen Rückgabewert

- Initialisierung von globalen Variablen (vor main.main):
  - Initialisierung (default)
  - init() Funktion (eine pro file)
  - single threaded (korrekte Ausführung)

Pointer vs. Values

- Automatische Dereferenzierung bei Methodenaufrufen, sowie bei Wertezuweisung
- Keine Pointerarithmetik
- Kein & Operator
- Vorteil: weniger Fehlerquellen

Inter-Faces

Interfaces

- Definition:
  - Ein Interface ist eine Menge von Methoden
  - ähnlich wie Java Interfaces
  - Beispiel

Embedding

- Interface Embedding:
  - Vereinigt Interfaces als neues Interface
  - „subclassing“ / „mehrfachvererbung“

- Struct Embedding:

```
type ReaderWriter struct {
  Reader
  Writer
}
```

Weitere Eigenschaften

- Generics – noch nicht implementiert
  - zu hohe Ansprüche an Laufzeitumgebung
- Keine Exceptions
  - kann auch mit Fallunterscheidungen gelöst werden
- Kein Überladen von Methoden
  - für die Entwickler als sinnlos eingestuft

Quelle

http://golang.org/