Go

sECuRE beim NoName e.V.

powered by LaTeX, of course

3. Dezember 2009

Inhalt

- Syntax, Datentypen
- Funktionen
- Fehlerbehandlung
- Typsystem
- Interfaces
- Goroutinen
- Channels
- Beispiele

Syntax

- Sehr an C angelehnt
- Keine spezielle Pointer-dereferenzierung
- Variablen-Deklarationen mit `var`-keyword, Typ hinter den Namen
- Semikolon kann beim letzten Statement weggelassen werden

```
fmt.Printf("Hello World\n");
var foo, bar int;
foo = 3;
fmt.Printf("bye\n")
```

Datentypen

- int in verschiedenen Ausführungen (uint32, uint64, int64, . . .)
- string (nicht null-terminiert), grundsätzlich UTF-8-enkodiert
- Arrays mit fester Größe
- Slices (Referenz auf Teile eines Arrays)
- Maps

```
var days = [7]string {"Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"};
var weekdays []string;
weekdays = days[0:5];

var foo map[string]string;
foo = make(map[string]string);
foo["bar"] = "bleh";
foo["baz"] = "moo";
```

Short Declaration

- Typ der Variable aus dem Ausdruck bestimmt
- praktisch, weil kurz

```
var a int;
a = 3;

// bewirkt dasselbe wie:
b := 3;
days := [7]string{"Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"};
for i, v := range(days) {
        fmt.Printf("Element %d ist %s\n", i, v);
}

// Auslassungen bei Zuweisungen
for _, v := range(days) {
        fmt.Printf("Element %s\n", v);
}
```

Funktionen

- beliebig viele Parameter und Rückgabewerte (!)

- anonyme Funktionen

```
func foo() { fmt.Printf("foo() was called\n") }
func foo() int { return 23 }
func add(a, b int) int { return a + b }

var f func();
f = func() { fmt.Printf("anonym\n") };

func add_mul(a, b int) (int, int) {
        return (a + b), (a * b)
}
```

Defer

- Lässt einen Befehl am Ende einer Funktion (vor dem return) ausführen
- Argumente werden bei der Definition des Defers evaluiert

```
// (io.ReadLine ist pseudo-code)
func readImage(filename string) (string, bool) {
        f := os.Open(filename);
        defer f.Close();
        firstLine := io.ReadLine("\n");
        if firstLine != "JPG" {
                fmt.Printf("Wrong file type\n");
                return "", false
        }
        contents := io.ReadAll(f);
        return contents, true;
}
```

Fehlerbehandlung

- keine Exceptions
- zusätzlicher Rückgabewert (genannt: , ok-idiom)
- auch beim Prüfen auf Existenz in einer Map etc.

```
f, err := os.Open("/dev/null");
if err != nil {
        fmt.Printf("/dev/null could not be opened: %v\n", err);
        os.Exit(1)
}

v, ok := m["foo"];
if ok {
        fmt.Printf("Element is there, value is %v\n", v);
}
```

Typsystem (1)

- structs, wie in C
- Großgeschriebene Namen werden exportiert, kleingeschriebene nicht
- anonyme Felder in structs, dadurch ableiten möglich

```
type Point struct {
        x, y float
}
type Vector struct {
        p Point;
        direction float
}

var p Point;
p.x = 3; p.y = 4;

var v Vector;
v.x = 3; v.y = 4; v.direction = 5;
```

Typsystem (2)

- Methoden auf structs

- Methoden von anonymen Feldern werden vererbt

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

fmt.Printf("Distance = %d\n", p.Abs());
fmt.Printf("Distance of a vector = %d\n", v.Abs());
```

Interfaces

- Interface = eine Menge an Methoden (keine Daten!)
- ein Typ kann ein Interface implementieren
- ein Typ kann mehr Methoden anbieten als im Interface, d.h. er kann mehrere Interfaces implementieren
- beim Typ muss man nicht angeben, welche Interfaces er implementiert

```
// man beachte: (*Point) ist nicht Teil des Interfaces!
type AbsInterface interface {
        Abs() float
}

var ai AbsInterface;

ai = p; // da Point die Methode Abs() beinhaltet
fmt.Printf(ai.Abs());
```

Goroutinen

- Goroutine = eine Funktion, die gleichzeitig im selben Adressraum läuft
- Goroutinen sind nich teuer und einfach zu nutzen
- Ähnlich wie & auf der Shell

```
func IsReady(what string, minutes int) {
        time.Sleep(minutes * 60);
        fmt.Printf("%s is ready\n", what);
}

go IsReady("tea", 6);
go IsReady("coffee", 2);
fmt.Printf("waiting...\n");

// Ausgabe:
waiting... (sofort)
coffee is ready (nach 2 Minuten)
tea is ready (nach 6 Minuten)
```

Channels

- Kommunikation zwischen verschiedenen Goroutinen
- bestimmter Typ nötig
- synchron (oder mit Buffer)

```
func PrintValues(ch chan int) {
        fmt.Println("Waiting for values..");
        var i int;
        i <- ch;
        fmt.Printf("read %d\n", i);
}

ch := make(chan int);
go PrintValues(ch);
ch <- 3;
```

Channels (2)

- `range` funktioniert mit Channels
- Funktionen, die Channels zurückgeben
- Channels schließen (beendet range)

```
func pump() chan int {
        ch := make(chan int);
        go func() {
                for i := 0; i < 20; i++ { ch <- i }
                close(ch)
        }();
        return ch
}

stream := pump();
for v := range(stream) {
        fmt.Println(v)
}
```

Beispiele

- simples mapreduce
- regexp
- gobot

EOF

Fragen?