Arbeitsblatt #3

Einführung in die Programmierung mit *Go*

10. März 2015

1 Bitoperationen

Untersuchen Sie die Semantik der Bitoperatoren. Verwenden Sie dazu `fmt.Printf` mit Formatierungsverb `%b` mit verschiedenen Parametern. Was bewirken die betreffenden Operatoren?

a. Die binären Operatoren >> und <<, z.B. `a << b`

b. Der unäre Operator ^, z.B. `^x`

c. Die binären Operatoren `&`, `|`, und `&^`

2 Interfaces

Sortieren Sie die Bevölkerungszahlen aus dem letzten Arbeitsblatt mit Hilfe der Go-Standardbibliothek. Sie können Ihren existierenden Code zur Extraktion der Zahlen ggf. nutzen oder sonst neuen Code schreiben.

a. Betrachten Sie die Funktion `sort.Ints` unter `http://golang.org/pkg/sort/` und verwenden Sie diese zur Lösung der Aufgabe.

b. Sortieren Sie nun Ländernamen und Bevölkerungszahlen zusammen. Transformieren Sie dazu die `CountryMap` in eine Struktur `struct{name string; size int}` und verwenden Sie die Funktion `sort.Sort()`. Dazu müssen Sie das Interface `sort.Interface` implementieren.

   Sie können die nötigen Informationen entweder der Dokumentation des Interfaces entehmen oder Ihre Lösung analog des ersten Beispiels in der Dokumentation von `http://golang.org/pkg/sort/` umsetzen.

3 Funktionen als Werte

Implementieren Sie eine Funktion `Fold(s, v, f)`, die drei Parameter nimmt:

- `s`: Einen `int`-Slice
- `v`: Eine `int`-Zahl
- `f(int, int)int`: Eine beliebige Funktion, die zwei `int`-Zahlen verknüpft und eine `int`-Zahl zurückliefert.

Wenn `len(s)` = 0, soll Ihre Funktion `v` zurückliefern.
Wenn `s` = `[]int{x0}`, soll Ihre Funktion `f(v, x0)` liefern.
Wenn `s` = `[]int{x0, x1}`, soll Ihre Funktion `f(f(v, x0), x1)` zurückliefern.
Wenn `s` = `[]int{x0, ..., xn}`, soll Ihre Funktion `f(...(f(v, x0), ..., xn)` zurückliefern.

a. Implementieren Sie die betreffende Funktion.

b. Testen Sie Ihre Funktion, indem Sie `v=0` setzen und eine Additionsfunktion als `f` übergeben. Ihre Funktion liefert nun also die Summe aller Elemente des Slices.

c. Testen Sie Ihre Funktion, indem Sie eine 'Maximum'-Funktion, die den größeren der beiden Parameter zurückgibt, als `f` übergeben. Ihre Funktion liefert nun also das größte Element des Slices.

4 Komplexe Zahlen

Berechnen Sie die Mandelbrot-Menge und schreiben Sie das resultierende Fraktal in eine `png`-Datei.

Die Mandelbrot-Menge ist die Menge aller komplexen Zahlen p mit der Eigenschaft, daß die Folge

$$\begin{aligned} c_0 &= 0 + 0i \\ c_{k+1} &= c_k^2 + p \end{aligned}$$

nicht divergiert, also für ein beliebig großes k der Betrag der komplexen Zahl c_k kleiner als 4 ist.

Die wesentlichsten nötigen Funktionen finden Sie in Abbildung 1.

In diesem Code ist die Funktion `main` nur für die Ausgabe zuständig; die Berechnung des Bildes selbst wird in der Funktion `Render` durchgeführt. Dabei ist `basePos` die Startposition für das Fraktal und `scale` ein Vergrößerungsfaktor; Sie können die gegebenen Werte beibehalten.

`Render` iteriert über alle Bildpunkte des zu erzeugenden Bildes und berechnet einen Helligkeitswert (`count`) für jeden Punkt. Dieser Wert wird dann in das Bild eingetragen. Was fehlt, ist die tatsächliche Berechnung des Wertes `k` an der Stelle `BERECHNUNG`.

Diese Berechnung soll die Zahl k bestimmen, ab der $|c_k| \geq 4$, also die Iterationszahl k, nach der die Reihe divergiert. Implementieren Sie die oben angegebene Reihe, so daß k in der Variablen `k` gespeichert wird.

Implementierungshinweise:

- Sie können nach der Berechnung von c_{k+1} die Variable c_k verwerfen, da sie nicht mehr benötigt wird; sie können also einfach den 'aktuellen' Wert von c in einer Variablen `c` speichern.
- Sie können `cmplx.Abs(c)` verwenden, um den Betrag einer komplexen Zahl zu berechnen.
- **Wichtig:** Wenn p in der Mandelbrotmenge ist, divergiert die Berechnung nicht. Sollten Sie $k = 255$ erreichen, können Sie die Iteration abbrechen.

```go
package main

import (
    "image"
    "image/color"
    "image/png"
    "math/cmplx"
    "os"
)

func main() {
    image := Render(complex(0, 0), 3.0)
    writer, err := os.Create("mandelbrot.png")
    if err != nil {
        panic(err)
    }
    defer writer.Close()
    error := png.Encode(writer, image)
    if error != nil {
        panic(err)
    }
}

const width  = 640
const height = 480

func Render(basePos complex128, scale float64) image.Image {
    rect := image.Rectangle{image.Point{-width/2, -height/2},
                            image.Point{width/2, height/2}}
    // Erzeuge 'Leinwand' img zum Zeichnen, in Groesse des Rechtecks:
    img := image.NewRGBA(rect)
    for y := rect.Min.Y; y < rect.Max.Y; y++ {
        for x := rect.Min.X; x < rect.Max.X; x++ {
            p := basePos + complex(float64(x) / width,
                                   float64(y) / height)
                                         * complex(scale, 0)
            k := 0

            // BERECHNUNG

            img.Set(x, y, color.Gray{uint8(k)})
        }
    }
    return img
}
```

Abbildung 1: Teillösung für Mandelbrotberechnung