Rekursion

14. November 2023

Inhalt

Einleitung

Was gibt diese Funktion für $n = 3$ aus?

```
func CountDown(n int) {
    if n <= 0 {
        return
    }
    fmt.Println(n)
    CountDown(n - 1)
}
```

Einleitung

Beispiel: Addition als Gleichungen spezifiziert

$$x + 0 = x$$
$$x + s(y) = s(x + y)$$

Anwendung der Gleichungen:

$$\begin{array}{rlr}
 & s(s(0)) + s(s(s(0))) & \\
s(& s(s(0)) + s(s(0)) &) \\
s(s(& s(s(0)) + s(0) &)) \\
s(s(s(& s(s(0)) + 0 &))) \\
s(s(s(& s(s(0)) &)))
\end{array}$$

Einleitung

Rekursive Addition als Go-Programm:

```
func Add1(x, y int) int {

    // Gleichungen für die Addition:
    // x + 0 = x
    // x + (y+1) = (x+y) + 1

    if y == 0 {
        return x
    }
    return Add1(x, y-1) + 1
}
```

Einleitung

Alternative Version (Tail-Recursion):

```
func Add2(x, y int) int {

    // Gleichungen für die Addition:
    // x + 0 = x
    // x + y = (x+1) + (y-1)

    if y == 0 {
        return x
    }
    return Add2(x+1, y-1)
}
```

Einleitung

Wozu Rekursion?

- Manches lässt sich kürzer und eleganter schreiben.
- Beispiel Fakultät:

$$fac(n) = \prod_{i=1}^{n} i \qquad \text{oder} \qquad \begin{aligned} fac(0) &= 1 \\ fac(n) &= n \cdot fac(n-1) \end{aligned}$$

- Als iteratives Go-Programm:

```
func FactorialIter(n int) int {
    result := 1
    for i := 2; i <= n; i++ {
        result *= i
    }
    return result
}
```

Einleitung

Wozu Rekursion?

- Manches lässt sich kürzer und eleganter schreiben.
- Beispiel Fakultät:

$$fac(n) = \prod_{i=1}^{n} i \qquad \text{oder} \qquad \begin{aligned} fac(0) &= 1 \\ fac(n) &= n \cdot fac(n-1) \end{aligned}$$

- Als rekursives `Go`-Programm:

```
func Factorial(n int) int {
    if n <= 1 {
        return 1
    }
    return n * Factorial(n-1)
}
```

Einleitung

Schema für rekursive Definitionen

- Ein oder mehrere Basisfälle (Rekursionsanfang, Anker).
- Ein oder mehrere rekursive Aufrufe (Rekursionsschritt).

Vergleich mit `while`-Schleifen

- Abbruchbedingung entspricht Rekursionsanfang.
- Schleifenrumpf entspricht Rekursionsschritt.

Beispiele

Beispiel: Fibonacci-Folge

$$\begin{aligned} fib(1) &= fib(2) = 1 \\ fib(n) &= fib(n-1) + fib(n-2) \end{aligned}$$

die ersten 10 Fibonacci-Zahlen:

n :	1	2	3	4	5	6	7	8	9	10
$fib(n)$:	1	1	2	3	5	8	13	21	34	55

Beispiele

Beispiel: Hailstone-Folge

- Beginne mit einer *natürlichen Zahl* n.
- Ist n gerade, so nimm als nächstes $n/2$.
- Ist n ungerade, so nimm als nächstes $3n+1$.
- Wiederhole, bis der Zyklus $4, 2, 1$ erreicht ist.

Beispiele:

$n = 1$: $1, 4, 2, 1$

$n = 2$: $2, 1, 4, 2, 1$

$n = 3$: $3, 10, 5, 16, 8, 4, 2, 1$

$n = 4$: $4, 2, 1$

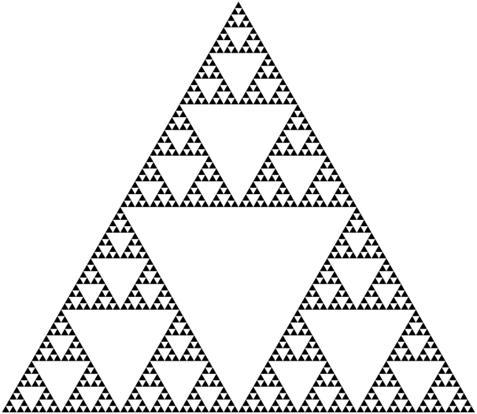
$n = 5$: $5, 16, 8, 4, 2, 1$

$n = 6$: $6, 3, 10, 5, 16, 8, 4, 2, 1$

$n = 7$: $7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$

Beispiele

Rekursion kann malen ...



Dieses Bild wird Sierpinski-Dreieck genannt.

Beispiele

Beispiel: Ackermann-Funktion

$$A(m,n) = \begin{cases} n+1 & \text{falls } m = 0 \\ A(m-1, 1) & \text{falls } m > 0 \text{ und } n = 0 \\ A(m-1, A(m, n-1)) & \text{falls } m > 0 \text{ und } n > 0 \end{cases}$$

Hintergrund

- Die Werte dieser Funktion wachsen extrem schnell!
- Die Funktion wurde erdacht, um zu beweisen, dass Schleifen ohne Laufzeitschranke beim Programmieren notwendig sind.
- Der Beweis hat die Wachstumsgeschwindigkeit der Ackermann-Funktion verwendet.

Türme von Hanoi

Aufgabe: Bewege einen Turm aus Spielsteinen von A nach C

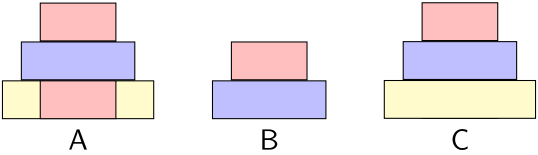
Gegeben:

- Spielsteine unterschiedlicher Größe.
- Drei Stellen A, B und C, an denen Spielsteine liegen können.

Spielregeln:

1. Die Steine werden einzeln bewegt.
2. Es wird niemals ein größerer Stein auf einen kleineren gelegt.

Beispiel mit 3 Steinen:



Türme von Hanoi

Frage: Wie bewegt man einen Turm der Höhe h von A nach C?

Naive Antwort:

1. Bewege alle bis auf die letzte Platte von A nach B
2. Bewege die letzte Platte von A nach C
3. Bewege den Turm von B nach C

Überraschung: So naiv ist das gar nicht!

- Wir konstruieren einen Algorithmus auf Basis dieser Vorgehensweise.

Türme von Hanoi

Wir definieren stückweise eine Funktion, die das Problem löst.

- Bewegen einer einzelnen Platte:

```
func Move(from, to string) {
    fmt.Printf("Bewege Scheibe
        von %s nach %s.\n", from,
        to)
}
```

Türme von Hanoi

Wir definieren stückweise eine Funktion, die das Problem löst.

- Bewegen eines Turms der Höhe 1:

```
func Hanoi1(from, via, to string)
    {
    Move(from, to)
}
```

Türme von Hanoi

Konstruktion der Hanoi-Lösung (Fortsetzung)

- Bewegen eines Turms der Höhe 2:

```
func Hanoi2(from, via, to string) {
    Hanoi1(from, to, via)
    Move(from, to)
    Hanoi1(via, from, to)
}
```

Türme von Hanoi

Konstruktion der Hanoi-Lösung (Fortsetzung)

- Bewegen eines Turms der Höhe 3:

```
func Hanoi3(from, via, to string)
    {
    Hanoi2(from, to, via)
    Move(from, to)
    Hanoi2(via, from, to)
}
```

Türme von Hanoi

Konstruktion der Hanoi-Lösung (Fortsetzung)

- Bewegen eines Turms der Höhe 4:

```
func Hanoi4(from, via, to string) {
    Hanoi3(from, to, via)
    Move(from, to)
    Hanoi3(via, from, to)
}
```

Laaaaaaaaa...

Türme von Hanoi

Konstruktion der Hanoi-Lösung (Fortsetzung)

- Bewegen eines Turms der Höhe 5:

```
func Hanoi5(from, via, to string) {
    Hanoi4(from, to, via)
    Move(from, to)
    Hanoi4(via, from, to)
}
```

...aaaaaaang...

Türme von Hanoi

Konstruktion der Hanoi-Lösung (Fortsetzung)

- Bewegen eines Turms der Höhe 6:

```
func Hanoi6(from, via, to string)
    {
     Hanoi5(from, to, via)
     Move(from, to)
     Hanoi5(via, from, to)
}
```

...weeeeilig

Türme von Hanoi

Beobachtungen:

- Die Funktionen `hanoi2`, `hanoi3`, `hanoi4`, …sind alle gleich.
- Beim Aufruf wird nur die Zahl reduziert und dann wieder das Gleiche gemacht.
- Nur bei `hanoi1` wird kein `hanoi0` aufgerufen.

Schlussfolgerung: Wenn die Höhe als Argument übergeben wird, können wir alles in eine Funktion schreiben.

Türme von Hanoi

Rekursive Hanoi-Lösung

```
func Hanoi(height int, from, via, to string) {
    if height == 1 {
        Move(from, to)
    } else {
        Hanoi(height-1, from, to, via)
        Move(from, to)
        Hanoi(height-1, via, from, to)
    }
}
```