

Go (Golang)

GOLANG



Peter Borovanský, KAI, I-18,
borovan(a)ii.fmph.uniba.sk

Jazyky:

- 1991, **Python**, Guido van Rossum,
- 1995, **Ruby**, Yukihiro Matsumoto,
- 2003, **Scala**, Martin Odersky,
- 2009, **Go**, Rob Pike, Ken Thompson

<http://www.python.org/>

<http://www.ruby-lang.org/en/>

<http://www.scala-lang.org/>

<http://golang.org/>

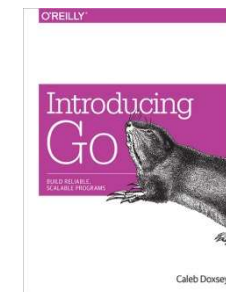
AN INTRODUCTION TO
PROGRAMMING
IN GO



CALEB DOXSEY

Literatúra:

- <https://www.golang-book.com/>
- <http://www.golangbootcamp.com/book/>
- <https://github.com/golang/go/wiki>
- <http://golang.org/ref/spec> - špecifikácia jazyka
- <http://talks.golang.org/2010/ExpressivenessOfGo-2010.pdf>
- <http://www.abclinuxu.cz/clanky/google-go-1.-narozneniny> (česky)



IDEs and Plugins

for Go

<https://github.com/golang/go/wiki/IDEsAndTextEditorPlugins>

LiteIDE (X36.1)

<http://liteide.org/en/>

GoLand od IntelliJ (2020.3)

<https://www.jetbrains.com/go/nextversion/>

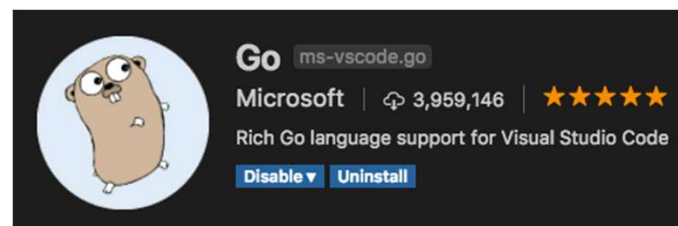
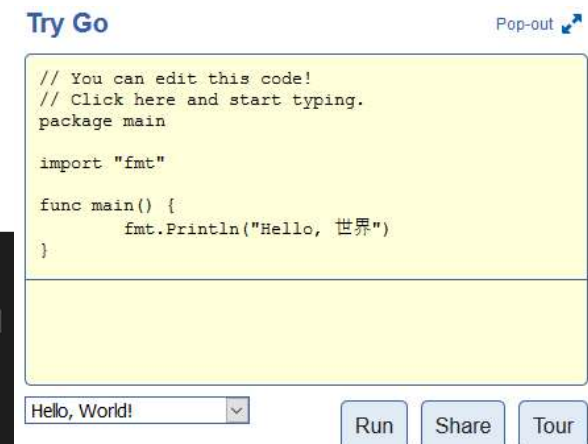
Online/repl/tutorials na

<https://repl.it/>

<https://golang.org/>

VSCode

<https://code.visualstudio.com/docs/languages/go>





Prečo Go

- procedurálny, **staticky a striktné typovaný** jazyk (ako Java)
- **poskytuje** možnosti/**pohodlie** dynamicky typovaných jazykov, (ako JavaScript, Python, Ruby, ...)
- je natívne **kompilovaný** (žiadna virtuálna mašina)
- je objektový, ale **nepozná** (pod)**triedy**, abstraktné metódy ani dedičnosť
- podporuje tzv. **implicitný interface** (ak objekt má predpísané metódy)
- **nepodporuje preťažovanie** (metód ani operátorov)
- *zatiaľ* (Go-1) nepodporuje generics/templates, Go-2 už roky v nedohľadne...
- ... má len "metódy", ale aj pre základné typy (int, string, float, ...)
- podporuje **funkcionálnu paradigmu**, podobne ako Lisp, Python, či Haskell
- ale hlavne podporuje **konkurentnú paradigmu**
- nemá predprocesor
- má **garbage collector** (... aj pre konkurentné rutiny nazývané gorutiny)
- nemá hlavičkové súbory, viditeľnú informáciu z modulu exportuje do .a

Year	Winner
2018	🏆 Python
2017	🏆 C
2016	🏆 Go
2015	🏆 Java
2014	🏆 JavaScript
2013	🏆 Transact-SQL
2012	🏆 Objective-C
2011	🏆 Objective-C
2010	🏆 Python
2009	🏆 Go
2008	🏆 C

TIOBE Programming Community Index Oct 2018

Sep 2020	Sep 2019	Change	Programming Language	Ratings	Change
1	2	▲	C	15.95%	+0.74%
2	1	▼	Java	13.48%	-3.18%
3	3		Python	10.47%	+0.59%
4	4		C++	7.11%	+1.48%
5	5		C#	4.58%	+1.18%
6	6		Visual Basic	4.12%	+0.83%
7	7		JavaScript	2.54%	+0.41%
8	9	▲	PHP	2.49%	+0.62%
9	19	▲▲	R	2.37%	+1.33%
10	8	▼	SQL	1.76%	-0.19%
11	14	▲	Go	1.46%	+0.24%
12	16	▲▲	Swift	1.38%	+0.28%
13	20	▲▲	Perl	1.30%	+0.26%
14	12	▼	Assembly language	1.30%	-0.08%
15	15		Ruby	1.24%	+0.03%
16	18	▲	MATLAB	1.10%	+0.04%
17	11	▼▼	Groovy	0.99%	-0.52%
18	33	▲▲	Rust	0.92%	+0.55%
19	10	▼▼	Objective-C	0.85%	-0.99%
20	24	▲▲	Dart	0.77%	+0.13%

36	Logo
37	LabVIEW
38	Prolog
39	Haskell
40	Scheme
41	Kotlin

<https://www.tiobe.com/tiobe-index/>

Prečo Go

Odporúčam prejsť článok Beauty of Go

<https://hackernoon.com/the-beauty-of-go-98057e3f0a7d>

Plusy:

- staticky typovaný jazyk
- rýchlosť kompilácie
- rýchlosť exekúcie programu
- portovaný na iné platformy (win, linux, freebsd, OS X)
- konkurencia na modeli Communicating Sequential Processes, Tony Hoare
- interfaces
- gargabge collection
- no exceptions handling - *do it yourself*

Mínusy:

- nemá parametrizované typy (roky sľúbené v Go-2)



Hello world

jednoduchosť a čitateľnosť

Prvý program v Go:

```
package main
import "fmt"
```

```
func main() {
    fmt.Println("Hello " + "world !")
}
```

// package je povinne 1.príkaz modulu

// spustiteľný package musí mať package main

// konvencia: package name je najhlbší podadresár

// fmt implementuje formátované I/O

// hlavná spustiteľná metóda main()

fmt.Println("Hello " + "world !") //viac na golang.org/pkg/fmt/

```
>go run hello.go
Hello world !
```

← spustenie v command line

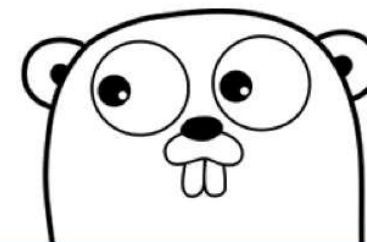
```
>go build hello.go
```

← kompilácia v command line

```
>dir hello.*
09/27/2019 05:30 PM
09/26/2019 04:38 PM
```

```
2,062,336 hello.exe
84 hello.go
```

```
>hello.exe
Hello world !
```



Hello/hello.go

Hello world



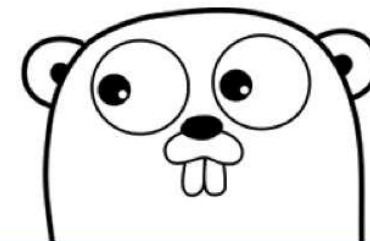
The screenshot shows a web browser window with the URL <https://repl.it/@mercury/helloW#main.go>. The interface includes a file explorer on the left with 'main.go' and 'go.mod' files. The main editor displays the following Go code in 'main.go':

```
1 package main
2 import "fmt"
3
4 func main() {
5     fmt.Println("Hello " + "world !")
6 }
7
```

A 'Run' button is visible above the editor. On the right, a terminal window shows the command execution:

```
https://helloW.mercury.repl.run
❯ go build
❯ ./main
Hello world !
❯
```

<https://repl.it/@mercury/helloW#main.go>



Hello/hello.go

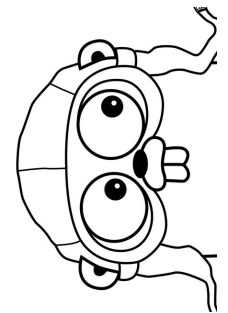
Základné typy a literály

orientovaný na reálny HW

Aby sme vedeli niečo programovať, potrebujeme aspoň základné dátové typy

- uint (uint8=byte, uint16, uint32, uint64)
- int (int8, int16, int32=rune, int64)
 - 28, 0100, 0xdeda, 817271910181011
- float (float32, float64)
 - 3.1415, 7.428e-11, 1E6
- complex (complex64, complex128)
 - 5i, 1.0+1i
- bool
 - true, false
- string
 - `hello` = "hello"
 - `\n` = "\\n\\n\\n"
 - "你好世界"
 - "\\xff\\u00FF"

// int = int32 alebo int64 podľa
// konkrétnej implementácie



// reťazec cez niekoľko riadkov

Operátory a pretypovanie

uznáva svet C++/Java

Priority pre binárne operátory:

- `*`, `/`, `%`, `<<`, `>>`, `&`, `&^` (bitový clear, t.j. and-xor)
- `+`, `-`, `|`, `^` (xor)
- `==`, `!=`, `<`, `<=`, `>`, `>=`
- `&&`
- `||`

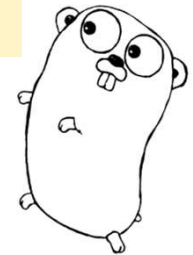
Unárne operátory:

- `^` bitová negácia int
- `!` not pre bool

Konverzie (ani medzi číselnými typmi) **nie sú implicitné**
syntax na pretypovanie `typ(výraz)`

- `float32(3.1415)` // 3.1415 typu float32
- `complex128(1)` // 1.0 + 0.0i typu complex128
- `float32(0.49999999)` // 0.5 typu float32

```
0b1100
&^ 0b1010
-----
0b0100
```



Premenné a ich deklarácie

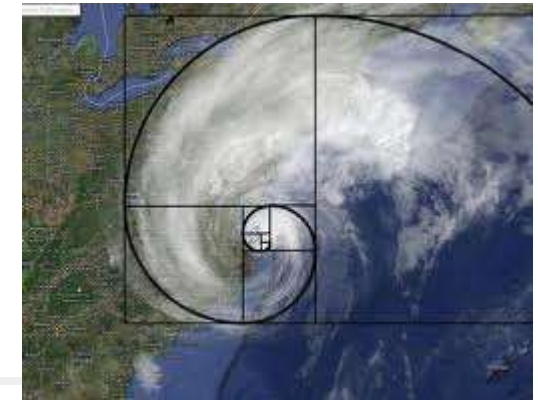
```
package main
import ("fmt" "strconv")
func main() {
    var hello string = "Hello"    // hello:string, bez :
    world := "world" // deklarácia world s inicializáciou
                          // jej typ sa inferuje z výrazu v pravo
    const dots = `...`           // konštanta typu string

    fmt.Println(hello + dots + world + strconv.Itoa(123))
    fmt.Println(hello + string(dots[0]) + world)
    str, err := strconv.Atoi("3.4") // str:string, err:Error
    if err == nil { fmt.Println(str) }
    else {
        fmt.Println("chyba: " + err.Error())
    }
}
```

- inferencia typu premennej pri deklarácii z výrazu inicializácie
- premenná = výraz priradí do premennej
- premenná := výraz deklaruje premennú



Fibonacci



V ďalšom ilustrujeme jazyk Go na triviálnom príklade

Fibonacciho postupnosti (Leonardo de Pisa, Pisano, 1170-1250)

1, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

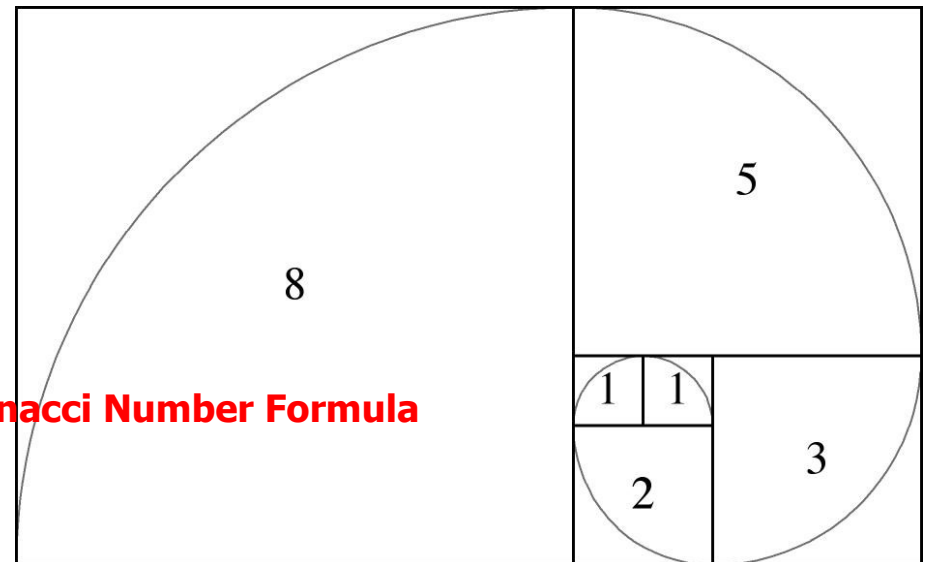
... exponenciálne rastie

Prejdeme:

- od triviálnej implementácie,
- cez tabelizáciu,
- výpočet pomocou veľkých čísel,
- logaritmicnú metódu,
- až po konkurentnú metódu
 - naivnú
 - efektívnu

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

Binet's Fibonacci Number Formula





Fibonacci a cyklus

- for výraz {...} je while
- for ...; ...; ... {...} podobne ako v Java
- Go má "paralelné priradenie"

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    var (
```

```
        a = 1
```

```
        b int = 1
```

```
        n int
```

```
    _, _ = fmt.Scanf("%d", &n)
```

```
    for ; n>0; n-- {
```

```
        fmt.Println(b)
```

```
        //a, b = a+b, a
```

```
        a = a+b
```

```
        b = a-b
```

```
    }
```

```
}
```

// viacnásobná deklarácia

// čítanie do n

// java-like for bez ()

// paralelné priradenie

```
>fibonacci
```

```
10
```

```
1
```

```
1
```

```
2
```

```
3
```

```
5
```

```
8
```

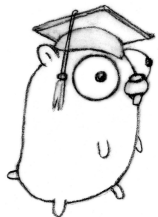
```
13
```

```
21
```

```
34
```

```
55
```

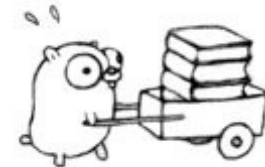
Fibo/fibCyklus.go



Fibonacci a rekurzia

func meno(argumenty) výsledný typ {...}

```
package main
import "fmt"
func Fib(n int) int {
    if n <= 2 {
        return 1
    } else {
        return Fib(n-2)+Fib(n-1)
    }
}
```



```
func main() {
    var n int
    _, _ = fmt.Scanf("%d", &n)
    for j:=1; j <= n; j++ {
        fmt.Println(Fib(j))
    }
}
```

// čítanie do n

Fibonacci a pole

typ pole je []element
Indexovanie 0 .. len()

```
var tabulka []int           // array[] of int, inic [0,0,0...]
func FibPole(n int) int {   // tabulka[i] = fib(i+1)
    if tabulka[n] == 0 {     // ak sme hodnotu este nepocitali
        tabulka[n] = FibPole(n-2) + FibPole(n-1) // pocitajme
    }                       // a zapamatajme do tabulky
    return tabulka[n]       // inak ju len vyberme z tabulky
}

func main() {
    var n int    _, _ = fmt.Scanf("%d", &n)
    tabulka = make([]int, n) // alokacia array[0..n-1] of int
    tabulka[0] = 1           // fib(1) = 1
    tabulka[1] = 1           // fib(2) = 1
    for j := 0; j < len(tabulka); j++ {
        fmt.Println(FibPole(j))
    }
}
```



```

type Int struct {
    neg bool        // sign
    abs []uintptr    // array of digits of a
                        multi-precision unsigned int.
}

```

Fibonacci a big

```

package main

import ( "fmt" "math/big" )

func FibBig(n int) *big.Int {
    if n < 2 {
        return big.NewInt(1)
    }
    a := big.NewInt(0)
    b := big.NewInt(1)
    for n > 0 { // while
        a.Add(a, b) // a = a+b
        b.Sub(a, b) // b = b-a
        n--
    }
    return a
}

```

pozri <http://golang.org/pkg/math/big/>

<https://www.wolframalpha.com/input/?i=fibonacci+1024>

 **WolframAlpha**

fibonacci 1024

Result:

450669961

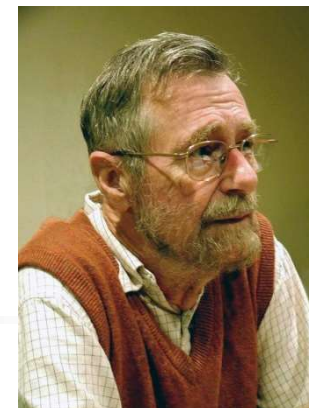
```

fibBig(1024) =
450669963367781981310438323572888604936786059621860483080
302314960003064570872139624879260914103039624487326658034
501121953020936742558101987106764609420026228520234665586
8899711089246778413354004103631553925405243

```

Fibo/fiboBig.go

Fibonacci logaritmicky



Základný hint pochádza odtiaľto (F_j je alias pre $\text{Fib}(j)$):

<http://www.cs.utexas.edu/users/EWD/ewd06xx/EWD654.PDF>

Eventually I found a way of deriving these schemes. for $k = 2$, the normal Fibonacci numbers, the method leads to the well-known formulae

$$F_{2j} = F_j^2 + F_{j+1}^2$$

$$F_{2j+1} = (2F_j + F_{j+1}) * F_{j+1} \quad \text{or} \quad F_{2j-1} = (2F_{j+1} - F_j) * F_j$$

!!! Domáca úloha: dokážte to (indukciou :-)

Logaritmická idea ako vypočítať F_j , resp. $\text{Fib}(j)$, pre veľké j !

- $F_{1024} = F_{0b10000000000}$
 $(F_{512}, F_{513}) < -(F_{256}, F_{257}) < -(F_{128}, F_{129}) < -(F_{64}, F_{65}) < -(F_{32}, F_{33}) < \dots (F_1, F_2) = (0, 1)$
... a ako bonus dostaneme aj F_{1025} :-)
- $F_{10} = F_{1010}$
 $(F_{10}, F_{11}) < -(F_5, F_6) < -??? (F_4, F_5) < -(F_2, F_3) < -(F_1, F_2) = (0, 1)$



Viac výstupov funkcie

`func meno(argumenty) (typ1, ... typN) {...}`

Funkcia môže mať viac výstupných hodnôt :-) :-) :-)

```
func FibPair(Fj int, Fj1 int) (int, int) {  
    return Fj*Fj + Fj1*Fj1, (Fj + Fj + Fj1) * Fj1  
}
```

$$\begin{aligned} F_{2j} &= F_j^2 + F_{j+1}^2 \\ F_{2j+1} &= (2 * F_j + F_{j+1}) * F_{j+1} \end{aligned}$$

```
func FibLog(n int) (int, int) {  
    // funguje pre n = 2^i  
    // return pre viac hodnôt  
    // F1 = 0, F2 = 1  
    if n < 2 {  
        return 0, 1  
    } else {  
        fj, fj1 := FibLog(n / 2) // výstup rekurzie (dvojicu)  
        return FibPair(fj, fj1) // môžeme priamo poslať  
    }  
} // return FibPair(FibLog(n/2))  
Fibo/fibLog.go
```



FibLog pre párne aj nepárne

Doriešime prípad, ak n nie je mocnina 2

`func meno(argumenty) (typ1, ... typN) {...}`

```
func FibLog(n int) (int, int) {  
    if n < 2 {  
        return 0, 1  
    }  
    else if n%2 == 1 { // pre n nepárne  
        x, y := FibPair(fibLog(n / 2))  
        return y, y + x  
    } else { // pre n párne  
        return FibPair(fibLog(n / 2))  
    }  
}
```

ako $(F_5, F_6) <^{???} (F_4, F_5)$

Idea:

$(F_5, F_6) = (F_5, F_5 + F_4)$

```
func FibLog1(n int) int { // vráť druhý z výsledkov  
    _, y := FibLog(n)    // neviem to šikovnejšie urobiť  
    return y             // snáď to niekto objaví ...  
}
```

Fibo/fibLog.go



FibLogBig

$$\begin{aligned} F_{2j} &= F_j^2 + F_{j+1}^2 \\ F_{2j+1} &= (2 * F_j + F_{j+1}) * F_{j+1} \end{aligned}$$

Jednoduchým spôsobom upravíme FibPair z int na *big.Int,

```
func FibPair(Fj int, Fj1 int) (int, int) {  
    return Fj*Fj + Fj1*Fj1, (Fj + Fj + Fj1) * Fj1  
modul math/big z.Add(x,y) je z = x+y, z.Mul(x,y) je z = x*y,  
detaily k math/big hľadať v http://golang.org/pkg/math/big/
```

```
func FibPairBig(Fj *big.Int, Fj1 *big.Int)  
    (*big.Int, *big.Int) {  
    tmp := new(big.Int)                // pomocná prem :: big  
    F2j1 := new(big.Int)               // - ... - F2j+1  
    F2j1.Mul(tmp.Add(tmp.Add(Fj, Fj), Fj1), Fj1)  
    F2j := new(big.Int)                // - ... - F2j  
    F2j.Add(Fj.Mul(Fj, Fj), Fj1.Mul(Fj1, Fj1))  
    return F2j, F2j1  
}
```



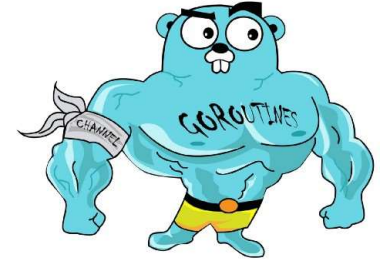
FibLogBig

Potom už logaritmický Fibonacci s veľkými číslami dostaneme priamočiaro z rekurzie nad int:

```
func FibLogBig(n int) (*big.Int, *big.Int) {  
    if n < 2 {  
        return big.NewInt(0), big.NewInt(1) // F1=0, F2=1  
    } else if n%2 == 1 {  
        x, y := FibPairBig(FibLogBig(n / 2))  
        return y, x.Add(y, x)  
    } else {  
        return FibPairBig(FibLogBig(n / 2))  
    }  
}
```

Go rutiny

(príklad z Java)



```
package main
```

go f(args) spustí extra vlákno na výpočet f()

```
import (    "fmt"    "math/rand"    "time")
```

```
func f(n int) {
```

```
    for i := 5; i > 0; i-- {
```

```
        fmt.Println("#", n, ":", i)
```

```
        time.Sleep(
```

```
            time.Duration(rand.Intn(500))*time.Millisecond)
```

```
    }}
```

```
func main() {
```

```
    for i := 0; i < 5; i++ {
```

```
        go f(i)
```

```
    }
```

```
    var input string
```

// toto čaká na input, v opačnom

```
    fmt.Scanln(&input)
```

// prípade, keď umrie hlavné

```
    fmt.Println("main stop") } // vlákno, umrú všetky ostatné
```

Concurrency/main.go

Do not communicate by sharing memory;
instead, share memory by communicating.



Kanály

- `make(chan int)` // nebufrovaný kanál int-ov
- `make(chan *big.Int, 100)` // bufrovaný kanál veľkosti 100 pre *big.Int

Nebufrovaný kanál `ch` typu `chan e` umožňuje

- komunikovať medzi go-rutinami,
 - `ch <- val` je synchronizovaný zápis do kanálu, kde `val` je typu `e`
 - `val <- ch` je synchronizované čítanie z kanála, kde `val` je typu `e`

My použijeme jednoduchý pattern:

```
ch := make(chan int)           // vytvorí nebufrovaný kanál s int-ami
go PocitajNieco(..., ch)       // spustí nezávislé vlákno
...                               // hlavné vlákno počíta ďalej
vysledok := <-ch                // čaká na výsledok, blokujúca operácia

func PocitajNieco(..., ch chan int) {
    trápenie, veľa trápenia, ..., inferno ...
    ch <- vysledok               // koniec ťažkého výpočtu, pošle do ch
                                // tiež blokujúca operácia, kým to neprežítá
}
```



FibPara

```
func FibPara(n int, ch chan int) {  
    if n < 2 {  
        ch <- 1  
    } else {  
        ch1 := make(chan int)  
        go FibPara(n-2, ch1)  
        ch2 := make(chan int)  
        go FibPara(n-1, ch2)  
        n1 := <-ch1           // čakáme na Fib(n-2)  
        n2 := <-ch2           // čakáme na Fib(n-1)  
        ch <- n1 + n2         // spočítame a kanalizujeme  
    }  
}
```

„Elegantný“, navyše konkurentný, kód... ale ???

Ale, zamyslime sa, koľko vlákien sa vytvorí na výpočet, napr. Fib(20) ???

Prívetá vlákien



```
ch := make(chan int)
go FibPara(20, ch)
res := <-ch
fmt.Printf("FibPara(20) %v\n", res)
```

Koľko vlákien to vygeneruje (kvíz) ??

- 10
- 100
- 1000
- 10000

Podme teda radšej paralelizovať viaceré násobenia veľkých čísel:

Fib(10^5) má cca 25tis. cifier, násobenie takých čísel nie je elementárna operácia

```
func multiplicator(a *big.Int, b *big.Int, ch chan *big.Int) {
    tmp := new(big.Int)
    ch <- tmp.Mul(a, b)
}
```

Fibo/fibPara.go



Paralelizuj s rozvahou

```
func FibPairBigPara(Fj *big.Int, Fj1 *big.Int) (*big.Int,
    *big.Int) {
    tmp := new(big.Int)
    tmp.Add(tmp.Add(Fj, Fj), Fj1)
    ch1 := make(chan *big.Int)
    go multiplicator(tmp, Fj1, ch1) // spusti 1.násobenie v 1.vlákně
    F2j := new(big.Int)
    ch2 := make(chan *big.Int)
    go multiplicator(Fj, Fj, ch2) // spusti 2.násobenie v 2.vlákně
    ch3 := make(chan *big.Int)
    go multiplicator(Fj1, Fj1, ch3) // spusti 3.násobenie v 3.vlákně
    F2j.Add(<-ch2, <-ch3) // čakaj na 2. a 3. výsledok
    return F2j, <-ch1 // a potrebuješ aj 1.výsledok
}
```

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix}$$

Fibonacci a matice

!!! A opäť domáca úloha:
dokážte to (indukciou :-)

```
package main

type Matrix struct {           // struct ako v C++
    a11, a12, a21, a22 int}    // reprezentácia matice 2x2
func (m *Matrix) multiply(n *Matrix) *Matrix {
    var c = &Matrix{           // konštruktor struct
        m.a11*n.a11 + m.a12*n.a21, // násobenie matíc 2x2,
        m.a11*n.a12 + m.a12*n.a22, // hardcoded-žiadne cykly
        m.a21*n.a11 + m.a22*n.a21,
        m.a21*n.a12 + m.a22*n.a22}
    return c                    // vráti pointer na maticu
}                               // teda Go má pointre, operátory &, * skoro ako C++
func FibMatrix(n int) int {
    m := &Matrix{a11: 1, a12: 1, a21: 1, a22: 0}
    p := m.power(n)
    return p.a12}

```



Logaritmický power

A už len to nezabiť tým, že power(), alias mocninu urobíme lineárnu...
Lepšia bude logaritmická verzia

```
func (m *Matrix) power(n int) *Matrix {  
    if n == 1 {  
        return m  
    } else if n%2 == 0 { //  $m^n = (m^{n/2})^2$ , pre n párne  
        m2 := m.power(n / 2)  
        return m2.multiply(m2)  
    } else { //  $m^n = m * (m^{n-1})$ , pre n nepárne  
        return m.power(n - 1).multiply(m)  
    }  
}
```



Trúfame si

počítať veľký

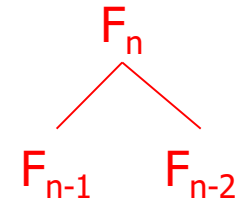
- faktoriál, napr. 1.000.000!
- koľko má cifier ?
- $\log(a*b) = \log a + \log b$
- $\log 1 + \log 2 + \dots + \log 1.000.000 = 5.565.709$ cifier...

- Kombinačné číslo $\binom{1\,000\,000}{500\,000} = 7.89957877227697084177023790317911268498339598305112 \times 10^{301026}$

- Partície n
- počet rôznych súčtov dávajúcich n
- partition 1.000.000 =

1 471 684 986 358 223 398 631 004 760 609 895 943 484 030 484 439 142 125 334 612 %
 747 351 666 117 418 918 618 276 330 148 873 983 597 555 842 015 374 130 600 288 %
 095 929 387 347 128 232 270 327 849 578 001 932 784 396 072 064 228 659 048 713 %
 020 170 971 840 761 025 676 479 860 846 908 142 829 356 706 929 785 991 290 519 %
 899 445 490 672 219 997 823 452 874 982 974 022 288 229 850 136 767 566 294 781 %
 887 494 687 879 003 824 699 988 197 729 200 632 068 668 735 996 662 273 816 798 %
 266 213 482 417 208 446 631 027 428 001 918 132 198 177 180 646 511 234 542 595 %
 026 728 424 452 592 296 781 193 448 139 994 664 730 105 742 564 359 154 794 989 %
 181 485 285 351 370 551 399 476 719 981 691 459 022 015 599 101 959 601 417 474 %
 075 715 430 750 022 184 895 815 209 339 012 481 734 469 448 319 323 280 150 665 %
 384 042 994 054 179 587 751 761 294 916 248 142 479 998 802 936 507 195 257 074 %
 485 047 571 662 771 763 903 391 442 495 113 823 298 195 263 008 336 489 826 045 %
 837 712 202 455 304 996 382 144 601 028 531 832 004 519 046 591 968 302 787 537 %
 418 118 486 000 612 016 852 593 542 741 980 215 046 267 245 473 237 321 845 833 %
 427 512 524 227 465 399 130 174 076 941 280 847 400 831 542 217 999 286 071 108 %
 336 303 316 298 289 102 444 649 696 805 395 416 791 875 480 010 852 636 774 022 %
 023 128 467 646 919 775 022 348 562 520 747 741 843 343 657 801 534 130 704 761 %
 975 530 375 169 707 999 287 040 285 677 841 619 347 472 368 171 772 154 046 664 %
 303 121 315 630 003 467 104 673 818

Štruktúry a smerníky

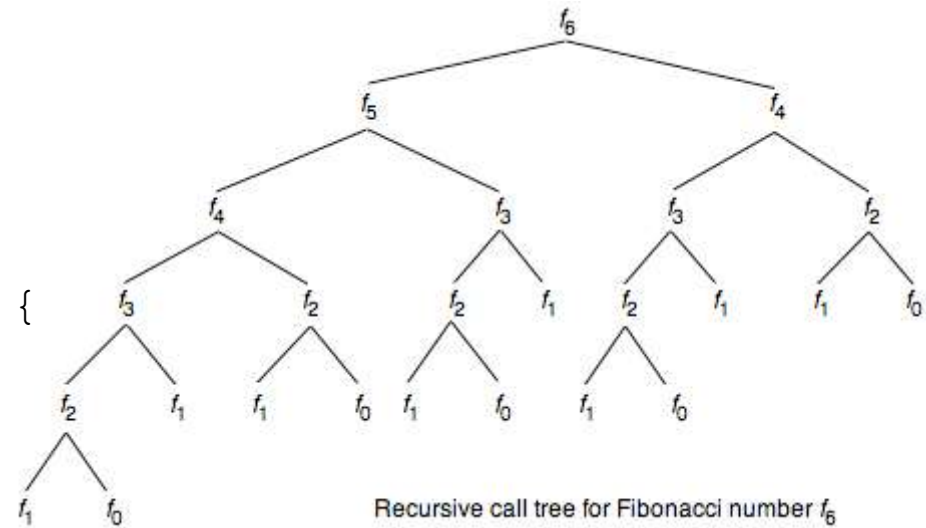


Program, ktorý generuje fibonacchiho strom definovaný štruktúrou:

```
type FibTree struct {
    left *FibTree
    right *FibTree
}

func generate(n int) *FibTree {
    if n < 2 {
        return nil
    } else {
        return &FibTree{generate(n - 1), generate(n - 2)}
    }
}

// alebo
// return &FibTree{left:generate(n-1),right:generate(n-2)}
// alebo
bt := new(FibTree) // alokuje krabicu pre FibTree
bt.left = generate(n - 1)
bt.right = generate(n - 2)
return bt }
```



Metódy

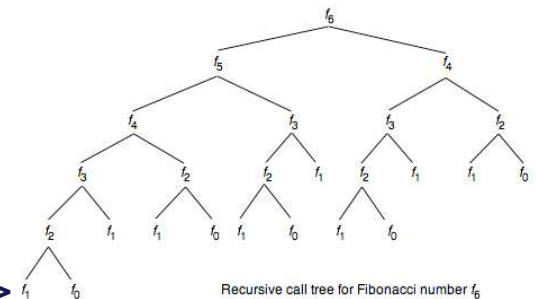
Nie je žiadne `this`, ani `self`, ako v iných jazykoch.

Parameter označujúci objekt, na ktorý sa metóda aplikuje, je explicitne prítomný v jej hlavičke, a to v zátvorkách **pred** menom metódy

```
func (bt *FibTree) size() int {
    if bt == nil { // metódu možno aplikovať na nil !!!
        return 1
    } else {
        return bt.left.size() + bt.right.size()
    }
}
```

}

size = 1 nil
 size = 1 nil
 size = 2 <nil,nil>
 size = 3 <<nil,nil>,nil>
 size = 5 <<<nil,nil>,nil>,<nil,nil>>
 size = 8 <<<<nil,nil>,nil>,<nil,nil>>,<<nil,nil>,nil>
 size = 13 <<<<<nil,nil>,nil>,<nil,nil>>,<<nil,nil>,nil>,<<<nil,nil>,nil>,<nil,nil>>>



Recursive call tree for Fibonacci number f_6



Logaritmický power

```
type realnaFunkcia /*=*/ func(float64) float64

func power(n int, f realnaFunkcia) realnaFunkcia {
    if n == 0 {
        return func(x float64) float64 { return x }
    } else if n%2 == 0 {
        // return kompozicia(power(n / 2, f), power(n / 2, f) )
        return func(x float64) float64 {
            rf := power(n / 2, f) // rf : realnaFunkcia
            return rf(rf(x))      //  $f^n = (f^{n/2}) \circ (f^{n/2})$ , pre n párne
        }
    } else {
        //  $f^n = f \circ (f^{n-1})$ , pre n nepárne
        // return kompozicia(f, power(n-1, f) )
        rf := power(n - 1, f) // rf : realnaFunkcia
        return func(x float64) float64 { return f(rf(x)) }
    }
}
```




Logaritmický power

(z cvičení)

```
type realnaFunkcia /*=*/ func(float64) float64

func power(n int, f realnaFunkcia) realnaFunkcia {
    if n == 0 {
        return func(x float64) float64 { return x }
    } else if n%2 == 0 {
        rf := power(n / 2, f)
        return kompozicia(rf, rf )
    } else {
        return kompozicia(f, power(n-1, f) )
    }
}
```



GO cheatsheet

■ <https://www.programming-idioms.org/cheatsheet/Go>

1 Print Hello World

Print a literal string on standard output

```
fmt.Println("Hello World")
```

2 Print Hello 10 times

Loop to execute some code a constant number of times

```
for i := 0; i < 10; i++ {  
    fmt.Println("Hello")  
}
```

3 Create a procedure

Like a function which doesn't return any value, thus has only side effects (e.g. Print to standard output)

```
func finish(name string) {  
    fmt.Println("My job here is done. Good bye " + name)  
}
```

4 Create a function which returns the square of an integer

```
func square(x int) int {  
    return x*x  
}
```

5 Create a 2D Point data structure

Declare a container type for two floating-point numbers *x* and *y*

```
type Point struct {  
    x, y float64  
}
```

6 Iterate over list values

Do something with each item *x* of an array-like collection *items*, regardless indexes.

```
for _, x := range items {  
    doSomething( x )  
}
```



Go helpovník

- **package & import**

```
package id // prvý riadok v module
import "path"
import ( "path" "path" ... ) // import viacerých
```

- **const, type, var, func**

```
const id ,... type = value, ... // deklarácia konštant s
                                explicitným typom
const id ,... = value ,... // deklarácia konštant s
                             implicitným typom
type id different_type // typové synonymum
var id ,... type = value ,... // deklarácia premenných
                              s explicitným typom
var id ,... = value ,... // deklarácia premenných
                          s implicitným typom
const|type|var ( spec; ... ) // viacnásobná deklarácia
```



Go heľpovník

- **if-then[-else]**

```
if [statement;] condition { block } [else if-or-block]
```

- **statement**

```
expression
```

// výraz, napr. 5*6

```
function call
```

// exp(2.71)

```
target ,... = expression ,...
```

// paralelné priradenie

```
target op= expression
```

// a += 5

```
target ++, target --
```

// !!! nie je to výraz

```
id ,... := expression ,...
```

// skrátená deklarácia

- **for cyklus**

```
for { block }
```

// for ;; {}, resp. while (true) {}

```
for condition { block }
```

// while

```
for init; condition; post { block }
```

// ako v Java

```
for index, value = range expression { block }
```

// cyklus cez pole, slice, kanál, ...



Go helpovník

■ arrays

```
type id [ length ] type           // typ poľa s konštantou dĺžkou
type id [ length ] [ length ] type // matica 2, 3-dimen.
len( array )                       // veľkosť poľa
expression [ expression ]         // indexovanie od 0..len(array) -
new([ length ] type )             // vytvorenie poľa s dĺžkou length
[ length ] type { expression , ... } // konštanta typu pole s dĺžkou
[ ] type { expression , ... }     // konštanta typu pole bez dĺžky
```

■ slices

```
[] type                           // slice type (without length)
make([] type, length)             // construction, slice and array
make([] type, length, capacity)
len( slice ), cap( slice )        // length and capacity
expression [low : high]          // construction from array or slice
expression [low :]
expression [ expression ]         // element, index from 0
```

Go heľpovník

- `function`

```
func id ( parameters ) { body } // procedúra
func id ( parameters ) result { body } // fcia s výstupným typom
func id ( parameters ) ( results ) { body } // viac výstupov
func ( parameters ) ( results ) { body } // anonymná funkcia
func ( types ) ( types ) // type // funkčný typ
id ,... type // parametre, výstupné premenné
id ... Type // neurčený počet argumentov
```

■ methods

```
func (id type) id ( parameters ) ( results ) { body }  
func (id *type) id ( parameters ) { body }           // definícia  
func ( type ,... ) ( results )                       // typ  
expression . id ( arguments )                        // volanie metódy  
type . id                                             // metóda ako funkcia  
(* type ). id
```



Go helpovník

■ **switch**

```
switch statement; expression {  
    case expression ,...: statement; ...  
    default: statement; ...  
}  
  
switch statement; id := expression .(type) {  
    case type ,...: statement; ...  
    default: statement; ...  
}
```

■ **maps**

```
var id map [ type1 ] type2 // deklarácia zobrazenia type1 -> type2  
make(map[type1 ] type2 ) // vytvorenie zobrazenia type1 -> type2  
make(map[type1] type2, initial-size)  
map [ key ] = value // pridanie (key -> value)  
map [ key ] = dummy, false // zmazanie key  
map [ key ] // vyhľadanie value pre key  
value, ok = map[ key ] // ok je true alebo false
```



Go helpovník

- **struct**

```
type id struct {  
    id ,... type // named field  
}
```

// deklarácia

- **new-make-nil**

```
new( type )  
make([] type, capacity )  
make([] type, length, capacity )  
make(chan type )  
make(chan type, capacity )  
make(map[ type ] type )  
make(map[ type ] type, initial-capacity )
```

// alokuje štruktúru a vráti *type

// pre slice, alokuje []type

// pre kanál, alokuje chan type

// pre map, alokuje map[type]type



FibBig(123456) has length 25801, time=165.0094ms

FibLogBig(123456) has length 25801, time=3.0002ms

FibLogBigPara(123456) has length 25801, time=5.0003ms

FibMatrixBig(123456) has length 25801, time=5.0003ms

FibLogBig(1234567) has length 258009, time=206.0118ms

FibLogBigPara(1234567) has length 258009, time=184.0105ms

FibMatrixBig(1234567) has length 258009, time=263.015ms

FibLogBig(12345678) has length 2580094, time=17.9820285s

FibLogBigPara(12345678) has length 2580094, time=17.0839772s

FibMatrixBig(12345678) has length 2580094, time=20.1881547s

FibLogBig(123456789) has length 25800943, time=50m24.5339934s

FibLogBigPara(123456789) has length 25800943, time=49m29.7948626s

FibMatrixBig(123456789) has length 25800943, time=51m36.7321229s



Rekordy 2019 (2x)

FibBig(123456) has length 25801, time=83.7793ms

FibLogBig(123456) has length 25801, time=1.9572ms

FibLogBigPara(123456) has length 25801, time=2.0199ms

FibMatrixBig(123456) has length 25801, time=3.0083ms

FibLogBig(1234567) has length 258009, time=124.6259ms

FibLogBigPara(1234567) has length 258009, time=111.738ms

FibMatrixBig(1234567) has length 258009, time=160.5684ms

FibLogBig(12345678) has length 2580094, time=11.2339694s

FibLogBigPara(12345678) has length 2580094, time=10.6081299s

FibMatrixBig(12345678) has length 2580094, time=12.5923135s

FibLogBig(123456789) has length 25800943, time=24m24.6244977s

FibLogBigPara(123456789) has length 25800943, time=24m57.1693336s

FibMatrixBig(123456789) has length 25800943, time=28m4.0541394s



Život v nevedomí

`#input.sk/struct2017/08.html`

```
import functools
import time
import sys
```

```
sys.setrecursionlimit(2*10**9)
```

```
@functools.lru_cache(maxsize=None)    # dekorátor
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

MemoryError: Stack overflow

```
print(*(fib(n) for n in range(10)))
t1 = time.perf_counter()
print(fib(12345))
t2 = time.perf_counter()
print('Seconds:', t2 - t1)
```

sys.setrecursionlimit(1010)**
OverflowError: Python int too large to
convert to C long



Rekordy.PY

FibIterativne(123456), time=3.1216s

FibIterativne(1234567), time=208s

Faktor 1000x ?

A to si ešte priznajme, že bigNums v .py sú napísané vlastne v C++

Epilóg:

Za vaše štúdium si možno kúpite 2 notebooky, nach každý bude 2x rýchlejší ako váš maturitný desktop, takže HW-speedup počas štúdia matfyzu je max. 4x

O to viac by ste mali premýšľať, ako ovplyvniť váš SW-speedup, najmä, ak o niečo ide

- kompilovať a nie interpretovať
- po prvej chodiacej verzii programu sa zamyslieť a prepísať ho efektívnejšie
- neignorovať matematiku, nie je to zlo programátora
- učiť sa algoritmy a chodiť na eaz
- chcieť poraziť wolframalpha.com