Functional Programming in Go

—

Dylan Meeus

whoami

@DylanMeeus

Tech lead @ dploy.ai

❤️ Go

❤️ Functional languages

One of these is not like the others..

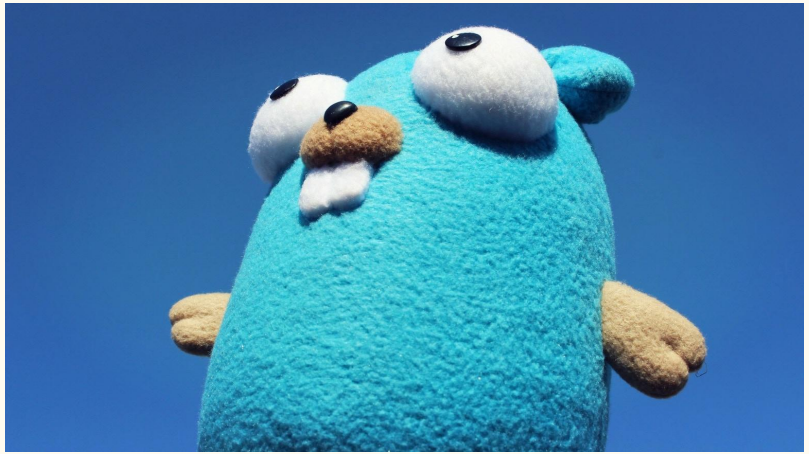
Haskell

Erlang

Go

Elm

Idris



When people think of FP...

Declarative

First-class functions

Pattern Matching

Pure functions

Immutability

...

Misconceptions?

Syntactic Sugar

Complicated

Never side-effects

Academic

...

Why FP?

Safer programs

Easier concurrency

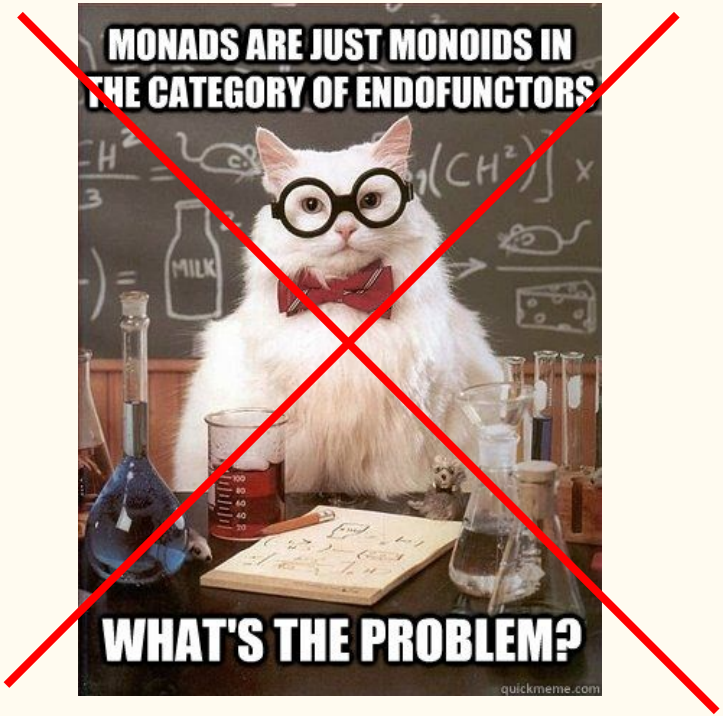
Less code, yet more expressive

Easier to debug & test

Fun?


Hmm interesting...

A gentle introduction



Code is:

- Declarative
- Immutable
- Pure
- Idempotent
- ‘Data’

Say what, not how

```
func Fun() int {
	sum := 0
	for i := -10; i <= 10; i++ {
		x :=
int(math.Abs(float64(i)))
		if x%2 == 0 {
			sum += x
		}
	}
	return sum
}
```

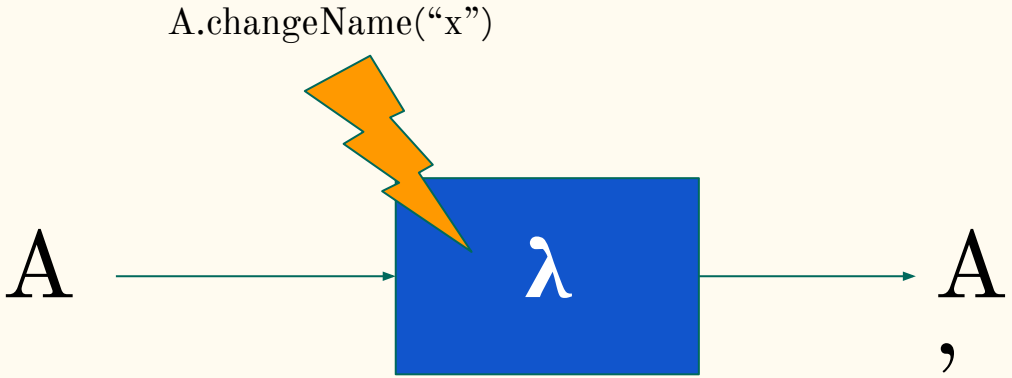
```
func Fun() int {
	return IntRange(-10, 10).
		Abs().
		Filter(func(i int64) bool {
			return i%2 == 0
		}).
		Sum()
}
```

Immutable functions

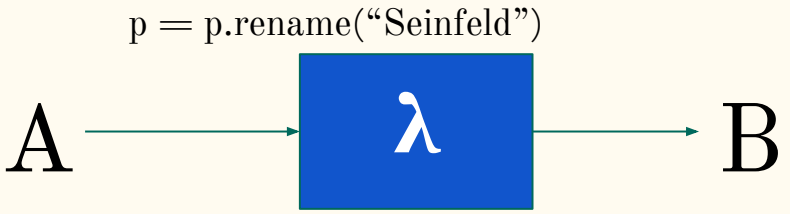
No state changes

Testable

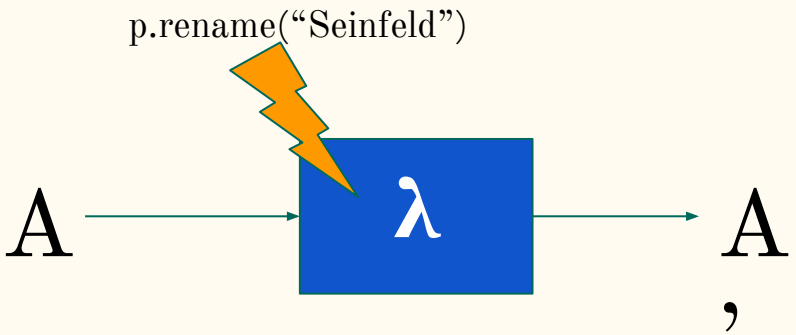
Safer concurrency



```
func (p *person) rename(s string) {
    p.name = s
}
```




```
func (p person) rename(s string) person {
    p.name = s
    return name
}
```



Pure function vs side-effect

```
func greet(s string) string {
     return "hello " + s
}
```

```
var callCounter int
func badGreet(s string) string {
     callCounter++
     return "hello " + s
}
```

```
func WriteFrames(samples []Frame, file
string) error {
     wfb := fmtToBytes(samples)
     hdr := createHeader(samples)

     bits := []byte{}
     bits = append(bits, hdr...)
     bits = append(bits, wfb...)

     return ioutil.WriteFile(file, bits,
0644)
}
```

Idempotence

```
func upperCase(s string) string {
     return strings.ToUpper(s)
}
```

```
func update(r row) row {
     r.lastModified = time.Now()
     return r
}

func update2(r row, t time.Time) row {
     r.lastModified = t
     return r
}
```

Functions always produce same result

You'll (probably) violate this somewhere

First-class / higher-order functions

```
func filter(ss []string, p func(string) bool) (out []string) {
	for _, s := range ss {
		if p(s) {
			out = append(out, s)
		}
	}
	return
}
```

```
func main() {
	s := []string{"hello", "world"}
	fmt.Println(filter(s, long))
}
```

```
func long(s string) bool {
	return len(s) > 10
}
```

Recursion vs Tail Recursion

```go
func fact(n int) int {
    if n == 1 {
        return 1
    }
    return n * fact(n-1)
}
```

```go
func tailFact(n int) int {
    return tailF(n-1, n)
}

func tailF(n, current int) int {
    if n == 1 {
        return current
    }
    return tailF(n-1, n*current)
}
```

Tail-Call Recursion

```
func tailFact(n int) int {
        return tailF(n-1,n)
}
```

```
func tailF(n, current int) int {
        if n == 1 {
            return current
        }
        return tailF(n - 1, n * current)
}
```

Why Tail-Recursion

Less stack-frame allocation

Can eliminate StackOverflows

No Optimization by Go compiler


+1!
LEVEL
UP

Up Next...

Closures

Currying

Real-World examples

Closure

```
func closure(s string) string {
     drop := func(i int) s
          return s[i:]
     }
     return drop(5)
}


func main() {
     fmt.Println(closure("hello world"))
     // prints -> world
}
```

inner function, referencing outer
guments

Closure

```
func closure(s string) string {
     drop := func(i int) string {
          return s[i:]
     }
     return drop(5)
}
```

```
func main() {
     fmt.Println(closure("hello world"))
     // prints -> world
}
```

inner function, referencing outer function arguments

Currying

Function with 1 parameter, returning Closure

```
func greet(p, n string) string {
     return fmt.Sprintf("%v %v\n", p, n)
}


func prefixGreet(p string) func(string) string {
     return func(n string) string {
          return greet(p, n)
     }
}
```

```
func main() {
     holaFn := prefixGreet("hola")
     helloFn := prefixGreet("hello")
     fmt.Println(holaFn("Gophers"))
     fmt.Println(helloFn("Gophers"))
     //“Hola Gophers” / “Hello Gophers”
}
```

Currying

Function with 1 parameter, returning Closure

```
func greet(p, n string) string {
   return fmt.Printf(“%v %v\n”, p, n)
}
```

```
func prefixGreet(p string) func(string) string {
    return func(n string) string {
        return greet(p, n)
    }
}
```

```
func main() {
    holaFn := prefixGreet(“hola”)
    helloFn := prefixGreet(“hello”)
    fmt.Println(holaFn(“Gophers”))
    fmt.Println(helloFn(“Gophers”))
}
```

Currying

Function with 1 parameter, returning Closure

```
func greet(p, n string) string {
   return fmt.Printf(“%v %v\n”, p, n)
}
```

```
func prefixGreet(p string) func(string) string {
    return func(n string) string {
        return greet(p, n)
    }
}
```

```
func main() {
    holaFn := prefixGreet(“hola”)
    helloFn := prefixGreet(“hello”)
    fmt.Println(holaFn(“Gophers”))
    fmt.Println(helloFn(“Gophers”))
}
```

Currying

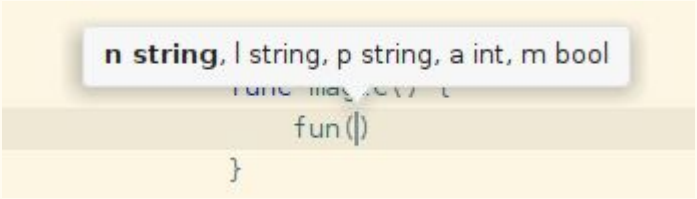
Function with 1 parameter, returning Closure

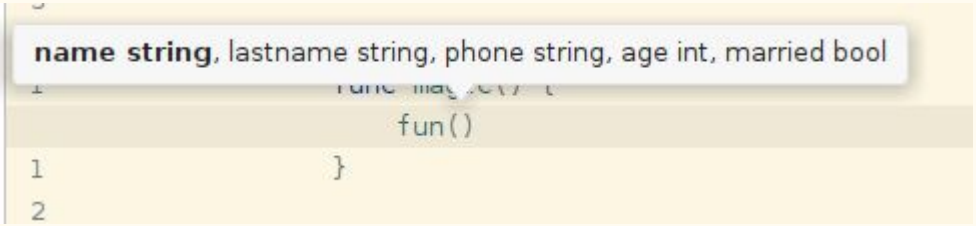
```
func greet(p, n string) string {
  return fmt.Printf("%v %v\n", p, n)
}
```

```
func prefixGreet(p string) func(string) string {
   return func(n string) string {
      return greet(p, n)
   }
}
```

```
func main() {
   holaFn := prefixGreet("hola")
   helloFn := prefixGreet("hello")
   fmt.Println(holaFn("Gophers"))
   fmt.Println(helloFn("Gophers"))
}
```

Practical use-case: Reduce function params





Server Options Example

```
type Server struct {
    opts options
}
```

```
type ServerOption func(o options) options
```

```
type options struct {
    maxCon        int
    transportType transport
    timeout       int
}
```

Defining Options

```
func MaxCon(n int) ServerOption {
    return func(o options) options
{
        o.maxCon = n
        return o
    }
}
```

```
func Timeout(n int) ServerOption {
    return func(o options) options
{
        o.timeout = n
        return o
    }
}
```

Constructor

```
func NewServer(os ...ServerOption) Server {
    opts := options{}

    for _, o := range os {
        opts = o(opts)
    }

    return Server{opts: opts}
}
```

```
func main() {
    s := NewServer(MaxCon(8), TransportType(UDP))
    ...
}
```

//maxCon: 8

//transport type: UDP

//timeout: 0

Default Options

```
func NewServer(os ...ServerOption) Server {
    opts := options{timeout: 1000, maxCon: 4}

    for _, o := range os {
        opts = o(opts)
    }

    return Server{opts: opts}
}
```

```
func main() {
    s := NewServer(MaxCon(8), TransportType(UDP))
    ...
}
```

//maxCon: 8

//transport type: UDP

//timeout: 1000

Declarative Programming

```
func filter(ss []string, p func(string) bool) (out []string) {
      for _, s := range ss {
            if p(s) {
                  out = append(out, s)
            }
      }
      return
}
```

```
func main() {
      s := []string{"hello", "world"}
      fs := filter(s,long)
      fmt.Println(fs)
}
```

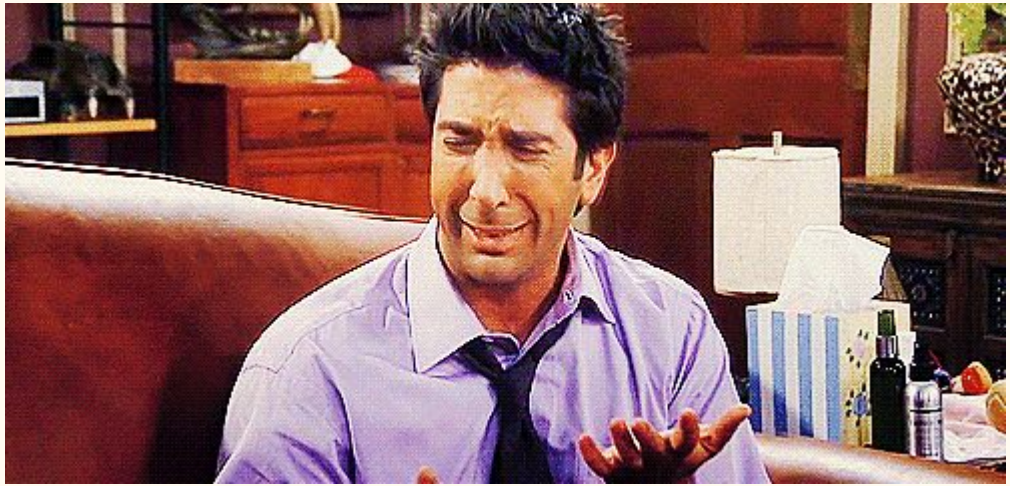
```
 func longStrings(s string) bool {
       return len(s) > 10
 }
```

Declarative Programming?

No generics (*yet)

Harder to maintain

Never entirely declarative



Libraries to the rescue

Gleam (Reflection) : https://github.com/chrislusf/gleam

Pie (Generator) : https://github.com/ElliotChance/pie

Hasgo (Generator) : https://github.com/DylanMeeus/hasgo

Hasgo

Generate Functions

Declarative

Pure

Nil-safe

Immutable

```
import (
	. "github.com/DylanMeeus/hasgo/types"
)


func Fun() int64 {
	result := IntRange(-10, 10).
		Abs().
		Filter(isEven).
		Sum()
	// result -> 60

}
```

Hasgo

Generate Functions

Declarative

Pure

Nil-safe

Immutable

```
func Fun() {
        result := IntRange(-10,10).
                        Abs().
                        Filter(isEven).
                        Sum()
        // result == 60
}
```

Hasgo

Generate Functions

Declarative

Pure

Nil-safe

Immutable

```
func Fun() {
        result := IntRange(-10,10).
                        Abs().
                        Filter(isEven).
                        Sum()
        // result == 60
}
```

Hasgo

Generate Functions

Declarative

Pure

Nil-safe

Immutable

```
func Fun() {
        result := IntRange(-10,10).
                        Abs().
                        Filter(isEven).
                        Sum()
        // result == 60
}
```

Hasgo

Generate Functions

Declarative

Pure

Nil-safe

Immutable

```
func Fun() {
        result := IntRange(-10,10).
                        Abs().
                        Filter(isEven).
                        Sum()
        // result == 60
}
```

Hasgo

Generate Functions

Declarative

Pure

Nil-safe

Immutable

```
func Fun() {
        result := IntRange(-10,10).
                        Abs().
                        Filter(isEven).
                        Sum()
        // result == 60
}
```

Generating functions

```
type Movies []Movie


type Movie struct {
    Name    string
    Owner   string
    Year    int
    Revenue int
}
```

Generating functions

```
//go:generate hasgo -T=Movie -S=Movies
type Movies []Movie


type Movie struct {
    Name    string
    Owner   string
    Year    int
    Revenue int
}
```

Generated Code

```go
// code generated by hasgo. [DO NOT EDIT!]
package types

import (
	"fmt"
)

// =============== all.go =================

// All returns true if all elements of the slice satisfy the predicate.
// Can be generated for any type.
func (s Movies) All(f func(Movie) bool) bool {
	if f == nil {
		return false
	}
	for _, v := range s {
		if !f(v) {
			return false
		}
	}
	return true
}

// =============== any.go =================

// Any returns true if any of the elements satisfy the predicate.
// Can be generated for any type.
func (s Movies) Any(f func(Movie) bool) bool {
	if f == nil {
		return false
	}
	for _, v := range s {
		if f(v) {
			return true
		}
	}
	return false
}
```

Code Generation Example

```
func GetMovies() Movies {
	return Movies{
		{
			Name:    "Star Wars: A New Hope",
			Owner:   "Lucasfilm",
			Year:    1977,
			Revenue: 10e8,
		},
		{
			Name:    "Toy Story",
			Owner:   "Pixar",
			Year:    1995,
			Revenue: 10e8,
		},
		… many more
	}
}
```

```
func makingMoney(m Movie) bool {
     return m.Revenue > 10e6
}
func buyRights(org string) Movies{
     mvs := GetMovies()
     res := mvs.Filter(makingMoney).
          Nub().
          Take(2).
          Map(func(m Movie) Movie {
               m.Owner = org
               return m
          })
     return res
}
```

```
func main() {
     buyRights("CartoonMouse")
}
```

And many more functions...

Abs
All
Any
Average
Break
Delete
Drop
DropWhile
Elem
Filter
Foldl
Foldl1
Foldr
Foldr1
Group
Head
Init
Inits
Intercalate
Intersperse
IsPrefixOf
Last
Length
Map
Maximum
MaximumBy
Minimum
Modes
Nub
Null
Product
Reverse
Scanl
Sort
Span
SplitAt
Sum
Tails
Take
TakeWhile
Uncons
Unlines
Unwords
IntRange
IntReplicate
Lines
StringReplicate
Words
And
Or

Add your own!

```
package functions

// TakeWhile appends to the output as long as the predicate is satisfied.
func (s SliceType) TakeWhile(p func(ElementType) bool) (out SliceType) {
	if len(s) == 0 {
		return
	}
	for _, e := range s {
		if !p(e) {
			return
		}
		out = append(out, e)
	}
	return
}
```

Some drawbacks..

Is it “Idiomatic Go”?

Never 100% Functional

Performance..sometimes

No syntactic sugar

Conclusion

It’s possible

Can improve code

Leverage libs

Thank You!

@DylanMeeus (Twitter, Github, Medium,...)

dylanmeeus.github.io

Hasgo: github.com/DylanMeeus/Hasgo